Univerza v Ljubljani
Fakulteta za računalništvo in informatiko

Domen Lanišnik

Razvoj mobilne aplikacije za Android v programskem jeziku Kotlin

diplomsko delo

visokošolski strokovni študijski program
prve stopnje
računalništvo in informatika

Mentor: doc. dr. Tomaž Dobravec

Ljubljana, 2018

Besedilo je oblikovano z urejevalnikom besedil LaTeX.

Fakulteta za računalništvo in informatiko izdaja naslednjo nalogo:

Tematika naloge:

V okviru diplomskega dela predstavite programski jezik Kotlin. Podrobneje opišite njegove zanimivejše konstrukte in ga primerjajte z Javo. Z uporabo Kotlina razvijte sistem za naročanje hrane preko mobilne aplikacije, ki bo restavracijam omogočal sprejemanje naročil. Sistem naj bo sestavljen iz spletnega servisa, mobilne aplikacije za uporabnike in spletne aplikacije za zaposlene. Opišite delovanje posameznega dela sistema in predstavite tehnologije in orodja, ki ste jih uporabili za razvoj.

Zahvaljujem se mentorju doc. dr. Tomažu Dobravcu za pomoč pri izdelavi diplomskega dela. Hvala vsem mojim bližnjim za vztrajno podporo na celotni študijski poti. Hvala tudi prijateljem, ki so študij naredili zanimivejši.

Kazalo

Seznam uporabljenih kratic

kratica	angleško	slovensko
CRUD	Create, read, update and delete	operacije za branje, ustvarjanje, spreminjanje in brisanje podatkov
CSS	Cascading Style Sheets	kaskadne stilske podloge
HTML	Hyper Text Markup Language	jezik za označevanje nadbesedila
HTTP	Hyper Text Transfer Protocol	hipertekstovni prenosni protokol
JPA	Java Persistence API	programski vmesnik za podatkovne baze v Javi
JSON	JavaScript Object Notation	oblika za izmenjavo podatkov
JVM	Java Virtual Machine	navidezni stroj Java
MVVM	Model–View–ViewModel	model-pogled-model pogleda
ORM	Object-relational mapping	objektno-relacijski preslikovalni mehanizem
POJO	Plain old Java object	običajen Java objekt
REST	Representational State Transfer	arhitektura za izmenjavo podatkov med spletnimi storitvami
URL	Uniform Resource Locator	enolični krajevnik vira
XML	Extensible Markup Language	razširljivi označevalni jezik

Povzetek

Naslov: Razvoj mobilne aplikacije za Android v programskem jeziku Kotlin

Avtor: Domen Lanišnik

Kotlin je sodoben programski jezik, ki poskuša nasloviti slabosti Jave, obenem pa ohraniti vse njene prednosti. Obljublja jedrnato in funkcionalno kodo z novimi koncepti, ki olajšajo implementacijo pogostih struktur. Njegova popularnost hitro narašča, še posebej pri razvoju mobilnih aplikacij za Android, kjer ni podprta uporaba najnovejših verzij Jave. Cilj diplomskega dela je bilo predstaviti programski jezik Kotlin in ga praktično uporabiti pri razvoju sistema za naročanje hrane preko mobilne aplikacije. Sistem je sestavljen iz spletnega servisa, mobilne aplikacije, ki uporabnikom omogoča enostavno oddajanje naročil, ter spletne aplikacije, ki zaposlenim v restavraciji omogoča upravljanje s prejetimi naročili. Spletni servis in spletno aplikacijo smo razvili v ogrodju Spring Boot, poleg Kotlina pa smo uporabili še tehnologiji HTML in CSS. Za shranjevanje podatkov smo uporabili relacijsko podatkovno bazo MySQL. Mobilno aplikacijo za Android smo razvili po priporočilih Googla, pri čemer smo uporabili arhitekturni vzorec MVVM.

Ključne besede: Kotlin, Android, mobilna aplikacija, spletni servis, Spring Boot.

Abstract

Title: Development of an Android mobile application in Kotlin programming language

Author: Domen Lanišnik

Kotlin is a modern programming language that tries to address some of the drawbacks of Java, while keeping all the advantages of Java. It promises concise and functional code with new concepts for easier implementation of common structures. Kotlin's popularity is growing fast, especially in the development of Android mobile applications, where the latest versions of Java are not supported. The goal of this thesis was to present Kotlin programming language and use it practically in the development of a mobile food ordering system. The system consists of a web service, a mobile application that enables users to easily create orders and a web application that enables restaurant employees to manage received orders. We developed the web application and web service in Spring Boot framework, using Kotlin and HTML and CSS technologies. For storing the data, we used MySQL relational database. We developed the Android mobile application according to Google guidelines, using MVVM architectural pattern.

Keywords: Kotlin, Android, mobile application, web service, Spring Boot.

Poglavje 1

Uvod

Programski jezik Java je že več let najpopularnejši jezik za razvoj aplikacij. Prva različica jezika je bila izdana leta 1996, trenutno pa je v uporabi deveta različica. Java ima kar nekaj prednosti pred ostalimi programskimi jeziki, med njimi so: enostavnost, preizkušenost, izvajanje programov na različnih operacijskih sistemih in najrazličnejši strojni opremi ter ogromna zbirka odprtokodnih orodij in knjižnic, ki so v pomoč razvijalcu. Poleg vseh prednosti ima Java tudi nekaj slabosti, kot sta naprimer upravljanje z ničelnimi vrednostmi in gostobesednost kode. Slabosti Jave poskuša nasloviti sodoben programski jezik Kotlin. Ta obljublja preprost in jedrnat programski jezik, z vsemi prednostmi Jave in novimi koncepti za lažjo implementacijo pogostih konstruktov. Njegova največja prednost je popolna interoperabilnost z Javo, kar pomeni, da lahko kodo napisano v Kotlinu kličemo tudi iz Jave in obratno.

Motivacija za predstavitev in uporabo programskega jezika Kotlin izhaja iz lastnih izkušenj pri razvoju mobilnih aplikacij za Android. Za razvoj se namreč uporablja okrnjena osma različica Jave, nekateri sodobnejši koncepti, kot so naprimer podatkovni tokovi, pa so podprti le na najnovejših verzijah operacijskega sistema. Zaradi želje po zmogljivejšem programskem jeziku in vse večje popularnosti Kotlina smo se odločili, da ga v diplomskem delu podrobneje predstavimo kot možno zamenjavo za Javo. Pri pregledu področja

smo ugotovili, da obstaja le eno delo, ki predstavi programski jezik Kotlin in je napisano v slovenščini. Gre za diplomsko delo [8], ki opisuje osnovne koncepte jezika in primerja hitrosti algoritmov urejanja v programskem jeziku Kotlin in Java. Medtem ko se omenjeno delo osredotoča predvsem na osnovne koncepte, pa naše delo opisuje tudi naprednejše koncepte in podrobneje primerja Kotlin in Javo. Cilj našega dela je na strukturiran in jasen način predstaviti nov programski jezik tistim, ki sedaj uporabljajo Javo in se želijo naučiti nov sodoben jezik.

V praktičnem delu diplomskega dela bomo v Kotlinu razvili prototip sistema za naročanje hrane, namenjen manjšim restavracijam, ki ponujajo dostavo hrane na dom, vendar naročila še vedno sprejemajo preko telefona. Pri takšnih naročilih lahko pride do napake zaradi slabe povezave ali nerazumevanja sogovornika. Z našo rešitvijo želimo postopek naročila olajšati tako strankam, kot tudi zaposlenim v restavraciji. Sistem bo sestavljen iz mobilne aplikacije, ki bo namenjena končnim uporabnikom oziroma strankam in spletne aplikacije, namenjene zaposlenim. Mobilna aplikacija bo uporabnikom na enostaven način omogočala pregled ponudbe restavracije in naročilo izbranih jedi na njihov naslov. Prijava v aplikacijo pa bo nadaljna naročila še olajšala. Prejeta naročila bo prikazovala spletna aplikacija, ki bo zaposlenim omogočala pregled naročil in spremljanje stanj naročil. Za povezovanje mobilne in spletne aplikacije pa bo skrbel spletni servis, ki bo dostopal do podatkov v podatkovni bazi.

Na trgu obstaja že nekaj podobnih rešitev, ki uporabniku omogočajo naročanje hrane preko mobilne ali spletne aplikacije. V Sloveniji je najpopularnejša ehrana, ki omogoča naročanje hrane iz večih restavracij. Ponudnikom dostave hrane na dom nudijo tudi aplikacijo za sprejem naročil, vendar si od vsakega izvedenega naročila vzamejo odstotek dobička. Naša rešitev je osredotočena le na eno restavracijo, kar uporabnikom olajša proces naročila, restavraciji pa nudi boljšo izpostavljenost in popoln nadzor nad celotnim sistemom.

Po uvodu in predstavitvi teme diplomskega dela bomo v drugem poglavju podrobneje predstavili programski jezik Kotlin. Po kratki zgodovini in primerjavi z Javo bomo predstavili njegove pomembnejše konstrukte. Najprej si bomo pogledali osnove jezika, kot so podatkovni tipi, varnost pred ničelnimi vrednostmi, razredi in funkcije. Nato bomo predstavili naprednejše konstrukte, kot so razširitvene funkcije, funkcije višjega reda, objekti in zapečateni razredi. Pri določenih konstruktih si bomo ogledali še njihovo implementacijo v ozadju.

V tretjem poglavju bomo predstavili sistem za naročanje hrane preko mobilne aplikacije in si podrobneje pogledali njegovo delovanje. Opisali bomo uporabljene tehnologije in potek razvoja mobilne aplikacije, spletne aplikacije, spletnega servisa in podatkovne baze.

V zadnjem poglavju bomo povzeli našo izkušnjo z uporabo programskega jezika Kotlin. Opisali bomo morebitne težave, s katerimi smo se srečali pri uporabi in predstavili konstrukte, ki so se izkazali za najbolj uporabne. Predstavili bomo tudi rezultate testiranja sistema za naročanje hrane. Na koncu bomo ocenili vrednost naše rešitve in podali možne izboljšave.

Poglavje 2

Kotlin

Programski jezik Kotlin [4] je razvilo podjetje JetBrains. Jezik so začeli razvijati za lastne potrebe leta 2010, leto kasneje pa so ga javno predstavili. Prva uradna verzija programskega jezika je bila izdana februarja 2016. V času pisanja (december 2017) je najnovejša verzija 1.2.10. Jezik je brezplačen in odprtokoden, izvorna koda pa je izdana pod licenco Apache 2.0.

Glavni namen Kotlina je ponuditi moderen jezik [7] tistim, ki sedaj uporabljajo Javo, saj ga lahko enostavno vključijo v že obstoječe projekte. Vključuje koncepte iz novejših programskih jezikov (kot sta naprimer C# in Swift) in poskuša nasloviti Javino gostobesedno kodo (angl. boilerplate code). Po ocenah razvijalcev jezika se število vrstic kode programa napisanega v Kotlinu, zmanjša za 40% [16] v primerjavi z istim programom napisanim v Javi.

Kotlin teče na navideznem stroju Java (angl. Java Virtual Machine, krajše JVM). JVM omogoča, da se preveden Java (ali Kotlin) program izvaja na različnih platformah. To zagotavlja prenosljivost Jave in s tem tudi Kotlina. Kotlin se tako kot Java prevede v zložno kodo (angl. Java bytecode), ki jo lahko označimo kot strojni jezik JVM. Te lastnosti omogočajo, da je Kotlin popolnoma interoperabilen z Javo, kar pomeni, da je lahko v enem projektu en del napisan v Javi, drugi pa v Kotlinu. Kotlin se lahko prevede v zložno kodo verzije 6 ali 8.

Maja 2017 je Google uradno podprl Kotlin kot enega izmed glavnih je-

zikov za razvoj aplikacij Android (poleg Jave in C++), s čimer je narasla popularnost jezika. Trenutno je za razvoj aplikacij Android podprta osma različica Jave, vendar pa so nekatere funkcionalnosti podprte le na novejših verzijah operacijskega sistema.

2.1 Osnove

2.1.1 Spremenljivke

Spremenljivko deklariramo tako, da najprej podamo ključno besedo `val` ali `var`, nato zapišemo ime spremenljivke, za dvopičjem pa še tip spremenljivke. Ključna beseda `val` pove, da se lahko vrednost spremenljivke le bere. V Javi to dosežemo z uporabo ključne besede `final`. Ključna beseda `var` pa se uporablja za običajne spremenljivke, ki jim lahko spreminjamo vrednosti. Pri deklaraciji spremenljivke lahko njen tip pogosto izpustimo, saj ga prevajalnik sam ugotovi na podlagi pripisane vrednosti.

```
1 val name: String = "Domen Lanisnik"
2 var age = 22
```

2.1.2 Podatkovni tipi

Java ima dve vrsti tipov spremenljivk [15]: osnovne ali primitivne (`int`, `long`, `boolean` ...) in sklicne tipe (`String`, polja ...). Za predstavitev primitivnih tipov kot objekte, Java uporablja tako imenovano ovijanje (angl. wrapping). V Kotlinu so vsi tipi tudi objekti. To pomeni, da lahko nad vsakim tipom kličemo funkcije in dostopamo do lastnosti. Javanski primitivni tipi so v času prevajanja preslikani v ustrezne tipe Kotlin, v času izvajanja pa se njihova predstavitev ne spremeni.

Numerični tipi

Kotlin pozna sledeče numerične tipe [15]: `Double`, `Float`, `Long`, `Int`, `Short` in `Byte`. Direktno pretvarjanje med tipi ni mogoče. Za pretvorbo se upora-

bljajo pomožne funkcije (`toInt()`, `toLong()` ...), ki jih ima vsak izmed tipov. Čeprav so vsi numerični tipi objekti, so na JVM predstavljeni kot osnovni tipi, razen v primeru ničelnih tipov in generikov se uporabi ovijanje.

```
1 val count: Long = 22L
2 val count1: Int = count // ni mogoce
3 val count2: Int = count.toInt()
```

Nizi

Nize [15], tako kot v Javi, deklariramo z dvojnimi narekovaji. Podprti so tudi večvrstični nizi, ki jih pišemo znotraj trojnih dvojnih narekovajev. Kotlin omogoča tudi interpolacijo nizov oziroma uporabo predlog. To olajša sestavljanje dinamičnih nizov in nadomešča konkatenacijo nizov v Javi. Znotraj niza lahko izpišemo vrednost neke spremenljivke, tako da pred njenim imenom dodamo simbol `$`. Na mestu spremenljivke se nato izpiše njena vrednost. Poleg uporabe spremenljivk lahko uporabljamo tudi izraze. Izraze zapišemo znotraj zavitih oklepajev.

```
1 val x = 10
2 println("Value of X is $x.") // "Value of X is 10."
3 println("Value of X-1 is ${x-1}.") // "Value of X-1 is 9."
```

Polja

Polja (angl. arrays) [15] so v Kotlinu predstavljena kot primerki razreda `Array<T>`. Za dostop do posameznega elementa polja se, tako kot v Javi, uporablja operator `[]`. Ustvarimo in inicializiramo jih lahko na dva načina.

Prvi način je uporaba pomožne funkcije `arrayOf()`, kateri za argumente podamo elemente polja, funkcija pa vrne ustvarjeno polje. Argumenti so lahko istega ali različnih tipov. Kot optimizacijo imamo na voljo tudi funkcije `booleanArrayOf()`, `shortArrayOf()`, `doubleArrayOf()`, `byteArrayOf()`, `intArrayOf()`, `longArrayOf()`, `charArrayOf()` in `floatArrayOf()`. Njihova posebnost je ta, da ne ustvarijo primerka razreda `Array<T>`, pač pa je polje na JVM predstavljeno kot polje osnovnih tipov. Še vedno pa lahko

nad tako ustvarjenim poljem kličemo vse funkcije, ki so na voljo pri ostalih. Večdimenzionalna polja ustvarimo tako, da za elemente uporabimo funkcijo `arrayOf()`.

Drugi način je uporaba konstruktorja `Array()`, ki zahteva velikost polja in funkcijo višjega reda (glej poglavje 2.5.2) za inicializacijo elementov. Inicializacijska funkcija sprejme indeks elementa in vrne vrednost, ki bo zapisana na tem indeksu. V spodnjem primeru tako ustvarimo polje kvadratov števil od 1 do 10.

```
1 val squares = Array(10, { e -> (e+1) * (e+1) })
2 // [1,4,9,16,25,36,49,64,81,100]
```

2.1.3 Zanke

Kotlin [21] podpira zanke `for`, `while` in `do-while`. Zadnji dve zanki sta sintaktično in funkcionalno enaki kot v Javi, medtem ko se zanka `for` v Kotlinu obnaša drugače. V resnici gre za Javino zanko `foreach`, saj lahko z njo iteriramo samo čez objekte, ki podpirajo iteracijo.

```
1 val numbers = arrayOf(5, 6, 7, 8)
2 for (e in numbers) {
3     println(e)
4 }
5 for (i in 1..20) {
6     println(i)
7 }
```

2.1.4 Odločitveni stavek

Stavek `if` [21] se od Javinega razlikuje po tem, da ni le stavek, temveč tudi izraz, kar pomeni, da vrne vrednost. Tako lahko neki spremenljivki določimo vrednost, ki jo izraz `if..else` vrne. Kotlin ne pozna pogojnega operatorja (angl. ternary operator), saj njegovo vlogo nadomešča običajni stavek `if..else`.

```
1 fun chooseBigger(x: Int, y: Int): Int {
2     return if (x > y) x else y
3 }
```

2.2 Varnost pred ničelnimi vrednostmi

V Javi ima lahko spremenljivka poljubnega tipa (razen primitivnih tipov) poleg dejanske vrednosti, tudi ničelno vrednost (angl. null). Ničelna vrednost lahko pomeni, da spremenljivka še ni bila inicializirana ali pa predstavlja posebno stanje, kot je naprimer napaka. Ta lastnost povzroči, da je treba pred uporabo spremenljivke ali klicem metode, prej preveriti ali je njena vrednost `null`. V primeru, da ima spremenljivka ničelno vrednost in poskušamo dostopati do njene lastnosti ali metode, se sproži izjema tipa `NullPointerException`. Težava pogosto nastane pri objektih, ki jih dobimo iz zunanjih virov, nad katerimi nimamo nadzora.

Kotlin [19] opisan problem rešuje tako, da loči med referencami, ki lahko imajo ničelno vrednost in tistimi, ki jo ne morejo imeti. Za razločevanje med spremenljivkami, ki lahko imajo ničelno vrednost, se uporablja simbol „`?`“, ki ga dodamo na konec tipa spremenljivke. Enako velja tudi za funkcije. Ničelno spremenljivko je ob deklaraciji treba obvezno inicializirati s privzeto vrednostjo, ki pa je lahko tudi `null`.

V spodnjem primeru najprej najavimo spremenljivki `a` in `b`. Spremenljivka `a` je tipa `String`, spremenljivka `b` pa tipa `String?`. Njuni vrednosti sta enaki. V naslednji vrstici poskušamo obema spremenljivkama nastaviti ničelno vrednost. V primeru spremenljivke `a` nas prevajalnik opozori, da leta ne dovoli ničelne vrednosti. Medtem ko lahko spremenljivki `b` brez težave nastavimo ničelno vrednost. V tretji vrstici poskušamo dostopati do dolžine niza. Pri spremenljivki `a` lahko dostopamo do dolžine, pri spremenljivki `b` pa nam prevajalnik javi napako, da dostop ni varen, saj ima lahko spremenljivka `b` ničelno vrednost.

```
1 var a: String = "Kotlin"
```

```
2  a = null // nedovoljen izraz
3  val lenA = a.length // ok
4
5  var b: String? = "Kotlin"
6  b = null // ok
7  val lenB = b.length // nedovoljen izraz
```

Če želimo dostopati do dolžine spremenljivke `b`, imamo na voljo več rešitev. Prva je poznana iz Jave, to je eksplicitno preverjanje za ničelno vrednost s pomočjo pogojnega stavka. Druga možnost je uporaba varnega operatorja „`?.`“. Ta vrne dolžino niza `b`, če vrednost niza ni `null`, v nasprotnem primeru pa vrne vrednost `null`. Na voljo nam je tudi tako imenovani operator „Elvis“ („`?:`“), ki združuje obe zgornji rešitvi. Če izraz levo od operatorja nima ničelne vrednosti, se uporabi vrednost izraza, drugače pa se uporabi rezultat izraza desno od operatorja. Pri tem velja, da je desna stran preverjena le v primeru, ko je vrednost leve strani enaka `null`. Varne operatorje lahko združujemo tudi v verige. To pomeni, da si lahko zaporedno sledi več operatorjev „`?.`“. Četrta možnost je operator „`!!`“. Ta bo vrnil neničelno vrednost spremenljivke `b` ali pa sprožil izjemo `NullPointerException`, če ima `b` ničelno vrednost.

```
1  val len: Int = if (b != null) b.length else -1
2  val len: Int? = b?.length
3  val len: Int = b?.length ?: -1
4  val trimLen: Int = b?.trim()?.length ?: 0
5  val len: Int = b!!.length
```

Ničelno spremenljivko lahko pretvorimo tudi v neničelni tip. Vendar pa se bo v primeru, da ima spremenljivka ničelno vrednost sprožila izjema `TypeCastException`. Kot rešitev obstaja tudi razširitvena funkcija `let`, ki se uporablja skupaj z varnim operatorjem, kadar nas zanima le neničelna vrednost. Programska koda znotraj funkcije se izvede le v primeru, ko je vrednost spremenljivke neničelna. Znotraj bloka `let` lahko do vrednosti varno dostopamo.

```
1  val b1 = b as String
```

```
2 b?.let {
3     val len: Int = b.length
4 }
```

Kotlin nam tako ponuja, da se sami odločimo ali ima lahko spremenljivka ničelno vrednost ali ne. S tem se izboljša razumljivost napisane programske kode in zmanjša verjetnost za izjemo `NullPointerException`. Kljub temu pa lahko do izjeme pride v naslednjih primerih:

- ko eksplicitno kličemo `throw NullPointerException()`,
- uporabimo operator „`!!`“ nad spremenljivko, ki ima ničelno vrednost,
- kličemo zunanjo javansko kodo, ki lahko povzroči izjemo.

2.3 Razredi

Razred [17] deklariramo s ključno besedo `class`, kateri sledi ime razreda in zaviti oklepaji, ki predstavljajo telo razreda. Nov primerek razreda ustvarimo podobno kot v Javi, le da izpustimo ključno besedo `new`.

2.3.1 Lastnosti

Za spremenljivko razreda uporabljamo izraz „lastnost“ (angl. property) [18]. Do lastnosti dostopamo neposredno, brez uporabe tako imenovanih metod „getter/setter“, kot je to priporočljivo v Javi. Če želimo za branje ali pisanje lastnosti uporabiti dodatno logiko, lahko to storimo z uporabo funkcij `get()` in `set()` v deklaraciji lastnosti. V spodnjem primeru se ob spremembi vrednosti lastnosti `title`, izpiše novo nastavljena vrednost.

```
1 class Movie {
2     val id: Long
3     var title: String
4         set(value) {
5             println(value)
6             field = value.capitalize()
```

```
7           }
8   }
9   val movie = Movie(1, "Forrest Gump", 142)
10  val id = movie.id
11  movie.title = "Casablanca"
```

2.3.2 Konstruktorji

Razred ima lahko en primarni konstruktor in enega ali več sekundarnih konstruktorjev. Primarni in sekundarni konstruktor [17] deklariramo z uporabo ključne besede `constructor`, ki pa jo lahko v nekaterih primerih tudi izpustimo. Primarni konstruktor je del glave deklaracije razreda in se nahaja takoj po imenu razreda. Konstruktor lahko vsebuje le parametre in ne inicializacijske kode. To lahko napišemo znotraj bloka `init`. Koda znotraj tega bloka se izvede takoj po kreiranju novega primerka razreda. Vrednosti lastnosti razreda lahko nastavimo znotraj bloka `init` ali pa direktno ob deklaraciji lastnosti. Omogočena pa je tudi deklaracija lastnosti razreda neposredno v konstruktorju. V tem primeru ni treba pisati inicializacijske kode.

```
1   class Movie(id: Long, title: String) {
2       val id: Long
3       val title: String = title
4
5       init {
6           this.id = id
7       }
8   }
9   // ali
10  class Movie(val id: Long, var title: String)
```

Kot že omenjeno, ima razred lahko tudi več sekundarnih konstruktorjev [17]. Sekundarni konstruktor mora obvezno klicati primarni konstruktor oziroma drug sekundarni konstruktor, ki kliče primarnega. Velikokrat je sekundarni konstruktor nepotreben, saj lahko parametrom v primarnem konstruktorju določimo privzete vrednosti (glej poglavje 2.4.2).

```
class Movie(val id: Long, var title: String) {
    constructor(title: String) : this(0, title) {
        println("Secondary constructor called")
    }
}
```

2.3.3 Dedovanje

Vsi razredi v Kotlinu imajo skupni nadrazred imenovan `Any` [17]. Če želimo dedovati iz nekega razreda, za imenom razreda dodamo dvopičje in nato ime dedovanega razreda. Razred, iz katerega dedujemo, mora dedovanje podpirati. Privzeto so vsi razredi v Kotlinu dokončni (angl. final), kar pomeni, da dedovanja ne podpirajo. Da lahko razred dedujemo, uporabimo pred deklaracijo razreda ključno besedo `open`. Prav tako je potrebno s ključno besedo `open` označiti vse funkcije in lastnosti, za katere želimo, da jih lahko povozimo v dedovanih razredih.

```
open class Person{
    var name = ""
    open fun printDescription(){
        println("$name is a person.")
    }
}

class Student: Person(){
    override fun printDescription() {
        super.printDescription()
        println("$name is a student")
    }
}
```

2.4 Funkcije

Funkcije [4] najavimo s ključno besedo `fun`, kateri sledi ime funkcije, nato pa v oklepajih podamo parametre. Tip vrednosti, ki jo vrne funkcija, najavimo

po parametrih. Če funkcija ne vrne eksplicitne vrednosti, bo njen tip `Unit`, ki ga pa ni treba najaviti posebej. Začetek in konec telesa funkcije določimo z zavitimi oklepaji. Spodnji primer prikazuje preprosto funkcijo, ki vrne seštevek dveh števil.

```
1 fun addNumbers(a: Int, b: Int): Int {
2     return a + b
3 }
```

2.4.1 Enovrstične funkcije

Funkcije [4], ki vsebujejo le en izraz, lahko krajše zapišemo tako, da zavite oklepaje izpustimo, ključno besedo `return` pa nadomestimo z znakom „=“. Poljubno lahko izpustimo tudi tip vrnjene vrednosti, če je le-tega mogoče ugotoviti s strani prevajalnika. Funkcijo, ki vrne seštevek dveh števil, bi tako krajše zapisali kot:

```
1 fun addNumbers(a: Int, b: Int) = a + b
```

2.4.2 Privzete vrednosti argumentov

Argumentom funkcije [4] lahko določimo privzete vrednosti, ki so uporabljene, če pri klicu funkcije izpustimo nek argument. To storimo tako, da po tipu parametra zapišemo še privzeto vrednost.

V spodnjem primeru imamo funkcijo za dodajanje novega filma v zbirko. Edini obvezen argument funkcije je naslov filma, medtem ko sta podnaslov filma in ali je film v barvni tehnologiji neobvezna argumenta in imata privzeti vrednosti.

```
1 fun addMovie(title: String, subtitle: String = "", colored:
      Boolean = true){/*implementacija*/}
```

Funkcijo lahko kličemo na več načinov, kot je to prikazano v naslednjem izseku kode. V prvi vrstici podamo le naslov filma, ostala dva argumenta pa izpustimo. Druga vrstica prikazuje spuščanje vmesnega argumenta, to je v našem primeru podnaslov filma. Ker pa lahko argumente izpuščamo

le od desne proti levi, je treba zadnji argument klicati po imenu. V primeru, da poimensko kličemo vse neobvezne argumente, pa njihov vrstni red ni pomemben. To je prikazana v tretji vrstici. Če pa se pri deklaraciji funkcije neobvezen argument nahaja pred obveznim, je uporaba poimenovanih argumentov pri klicu funkcije obvezna

```
1 addMovie("Interstellar")
2 addMovie("Casablanca", colored = false)
3 addMovie("Star Wars", colored = true, subtitle = "Episode V")
```

Java privzetih vrednosti argumentov ne podpira. V Javi je potrebno za podobno rešitev napisati preobložene metode (angl. overloaded methods) z različnim številom parametrov. V splošnem velja, da če ima metoda N argumentov in jih ima od tega M privzete vrednosti, je potrebnih M preobloženih metod. Prva metoda sprejme N-1 parametrov, naslednja N-2 in tako naprej.

Pri klicu Kotlin funkcije z opcijskimi parametri iz Jave je potrebno eksplicitno podati vrednost za vsak parameter. Če funkciji dodamo anotacijo `@JvmOverloads`, pa nam prevajalnik Kotlin generira vse preobložene metode, ki jih lahko nato kličemo iz Jave.

2.4.3 Funkcije najvišjega nivoja

Funkcije najvišjega nivoja (angl. top-level functions) [14] so funkcije, ki so definirane zunaj razreda, objekta ali vmesnika. Te funkcije lahko kličemo neposredno, kot da so del standardne knjižnice. Takšne funkcije definiramo v poljubni datoteki Kotlin, znotraj katerega koli paketa. Za dostop do funkcij v ostalih datotekah pa je potrebno funkcije uvoziti po imenu.

Primer: V datoteki `NumberUtils.kt` smo definirali funkcijo, ki vrne naključno število iz podanega intervala. Funkcijo smo nato klicali v datoteki `Functions.kt`, brez da bi se bilo potrebno sklicevati na določen objekt ali datoteko.

```
1 // Datoteka NumberUtils.kt
2 fun randomNumber(from: Int, to: Int): Int {
3     return Random().nextInt(to - from) + from
```

```
4 }
5
6 // Datoteka Functions.kt
7 fun main(args: Array<String>) {
8     println(randomNumber(10, 100))
9 }
```

Java takšnih funkcij oziroma metod ne podpira, saj morajo biti del razreda. Zato se v Javi pogosto uporabljajo statične metode znotraj pomožnih razredov. Ti pomožni razredi nimajo nobene dodatne logike, služijo le kot vsebniki za statične metode. Primer takšnega pomožnega razreda je na primer razred `Collections`.

Pri pregledu javanske kode, ki jo generira prevajalnik Kotlin, ugotovimo, da se v ozadju funkcije najvišjega nivoja pretvorijo v statične metode. Ustvari se nov razred, katerega ime je sestavljeno iz imena datoteke .kt in pripone „Kt“. Ustvarjen razred vsebuje statične metode. Tako se za funkcijo iz prejšnjega primera ustvari nov razred `NumberUtilsKt`.

```
1 public final class NumberUtilsKt {
2    public static final int randomNumber(int from, int to) {
3       return (new Random()).nextInt(to - from) + from;
4    }
5 }
```

Funkcije najvišjega nivoja, ki jih deklariramo v Kotlinu, kličemo iz Jave tako kot ostale statične metode: najprej podamo ime razreda v katerem se nahajajo, nato pa ime metode. Ime Java razreda, ki ga generira prevajalnik, lahko poljubno spremenimo z uporabo anotacije `@JvmName`. To deklariramo na začetku datoteke, pred imenom paketa, v katerem se datoteka nahaja. V oklepaju podamo želeno ime. Če bi torej želeli, da se generiran razred iz prejšnjega primera imenuje `NumberUtils`, bi na začetek datoteke `NumberUtils.kt` dodali anotacijo `@file:JvmName("NumberUtils")`.

2.4.4 Razširitvene funkcije

V Javi pogosto uporabljamo tako imenovane pomožne razrede (angl. utilility/helper class), s katerimi razširimo funkcionalnosti razredov nad katerimi nimamo vpliva (so naprimer del knjižnice, iz njih ne moremo dedovati, itd.). Ti pomožni razredi vsebujejo statične metode, ki za enega izmed parametrov prejmejo primerek razreda, ki mu želimo dodati novo funkcionalnost in nad njim izvedejo določene operacije.

Kot primer želimo nad nizi klicati funkcijo, ki bi obrnila vrstni red vseh črk v nizu. Iz razreda `String` ne moremo dedovati, saj je končen razred. Zato ustvarimo nov razred imenovan `StringUtils`, znotraj pa statično funkcijo `reverse`, ki kot argument prejme niz, ki ga želimo obrniti in vrne obrnjen niz.

Kotlin opisan problem rešuje s pomočjo razširitvenih funkcij (angl. extension functions) [1]. Te nam omogočajo, da nek razred razširimo z novimi funkcionalnostmi, brez potrebe po dedovanju tega razreda ali uporabe katerega izmed strukturnih vzorcev.

Razširitveno funkcijo lahko deklariramo zunaj razreda, nad katerim deluje ali pa kot del katerega drugega razreda. To pomeni, da gre za neko vrsto funkcije najvišjega nivoja. Pred ime funkcije dodamo ime razreda, ki ga razširjamo (angl. receiver type). Do primerka tega razreda lahko znotraj funkcije dostopamo s ključno besedo `this`. Takšno funkcijo lahko sedaj kličemo, kot da je del razreda, nad katerim deluje.

Prej opisan primer iz Jave lahko v Kotlinu realiziramo s pomočjo razširitvene funkcije. V novi datoteki imenovani `StringUtils.kt`, definiramo funkcijo `reverse()`, ki deluje nad razredom `String`. Sedaj lahko nad poljubnim nizom kličemo funkcijo `reverse()`, kot da je del razreda `String`.

```
1 // Datoteka StringUtils.kt
2 fun String.reverse(): String {
3     return StringBuilder(this).reverse().toString()
4 }
5
6 // Uporaba
```

```
7 println("Kotlin".reverse())
```

Pri pregledu javanske kode, ki jo generira prevajalnik Kotlin, ugotovimo, da se v ozadju ustvari statična metoda, ki kot prvi parameter prejme primerek razreda, nad katerim smo funkcijo poklicali [4]. V našem primeru se ustvari razred `StringUtilsKt`, ki vsebuje statično metodo `reverse()`, katero lahko v Javi kličemo na enak način, kot pri rešitvi opisani na začetku.

Razširitveno funkcijo lahko deklariramo tudi nad razredi ničelnega tipa. Takšne funkcije lahko varno kličemo tudi, če ima objekt ničelno vrednost. Primer takšne funkcije je funkcija `toString()`, ki izpiše opis objekta in jo lahko kličemo nad vsemi objekti v Kotlinu. Preverjanje za ničelno vrednost se zgodi znotraj funkcije.

```
1 fun Any?.toString(): String {
2     if (this == null)
3         return "null"
4     return toString()
5 }
```

Ena izmed omejitev razširitvenih funkcij je ta, da ne smemo povoziti funkcij, ki so že deklarirane v razredu. Če torej definiramo funkcijo z istim podpisom (isto ime funkcije, isto število, tip in vrstni red argumentov), je prevajalnik ne bo klical. V takem primeru se naprej poišče, če obstaja funkcija s podpisom, ki je del razreda, šele nato se preverijo razširitvene funkcije.

2.4.5 Infiksi zapis

Funkcijo, ki ima le en argument in je ali razširitvena funkcija ali pa pripada nekemu razredu, lahko deklariramo s predpono `infix` [14]. To funkcijo lahko nato kličemo brez uporabe pike, kot je to potrebno pri običajnih razrednih funkcijah. Infiksne funkcije nam omogočajo, da bolje izrazimo svoj namen. Kotlinova standardna knjižnica vsebuje nekaj uporabnih infiksnih funkcij. Primer takšne je funkcija `intersect`, ki izračuna presek dveh zbirk podatkov.

```
1 infix fun <T> Iterable<T>.intersect(other: Iterable<T>): Set<
      T>
```

```
2 // uporaba
3 val set1 = setOf(1, 2, 3, 4)
4 val set2 = setOf(3, 4, 5, 6)
5 val common = set1 intersect set2  // = [3, 4]
```

Če preverimo generirano zložno kodo, ugotovimo, da se infiksne funkcije pretvorijo v običajne metode.

2.5 Funkcije višjega reda

Funkcija višjega reda (angl. high-order function) [4] je funkcija, ki ima vsaj eno izmed sledečih lastnosti:

- za enega izmed parametrov prejme funkcijo (ali lambda izraz),
- za rezultat vrne funkcijo.

Če želimo, da funkcija za parameter prejme drugo funkcijo, je za ta parameter treba deklarirati funkcijski tip (angl. function type). Funkcijski tip opisuje podpis podprte funkcije. Znotraj oklepajev podamo parametre funkcije, nato sledi simbol „`->`“, za njim pa tip, ki ga funkcija vrne. Če torej želimo, da klicana funkcija sprejme celo število in vrne niz, to zapišemo kot `(Int) -> String`.

Funkcijo nato pošljemo kot parameter tako, da pred njenim imenom dodamo dve dvopičji, oklepaje in parametre pa izpustimo. Uporaba je prikazana na spodnjem primeru, kjer smo deklarirali funkcijo višjega reda `transform()`, ki prejme dva parametra: celo število in funkcijo, ki bo nad tem številom izvedla transformacijo in vrnila rezultat. Deklarirali smo tudi funkcijo `squared()`, ki prejme celo število in vrne njen kvadrat. Funkciji `transform()` smo nato pri klicu podali funkcijo `squared()`, kot je to prikazano v šesti vrstici. Pri klicu funkcije višjega reda lahko namesto druge funkcije, podamo tudi lambda izraz (več o lambda izrazih v poglavju 2.5.2). Uporaba lambda izraza je prikazana v osmi vrstici.

```
1 fun squared(number: Int): Int = number * number
```

```
2 fun transform(original: Int, function: (Int) -> Int): Int {
3     return function(original)
4 }
5
6 val result = transform(10, ::squared)
7 // ali z uporabo lambda izraza
8 val result = transform(10, { n -> n * n })
```

Funkcija lahko ne le sprejme drugo funkcijo kot parameter, temveč jo tudi vrne kot rezultat. To dosežemo tako, da funkciji za tip rezultata nastavimo funkcijski tip. Spodnji primer prikazuje deklaracijo funkcije `multiplier()`, ki vrne novo funkcijo. Vrnjeno funkcijo, ki pomnoži celo število s podanim faktorjem, shranimo v novo spremenljivko `multiplyBy5`. Spremenljivko lahko nato kličemo na isti način kot običajne funkcije.

```
1 fun multiplier(factor: Int): (Int) -> Int {
2     return { n -> n * factor }
3 }
4 val multiplyBy5 = multiplier(5)
5 val result = multiplyBy5(5) // 25
```

Tudi Java od verzije 8 najprej podpira funkcije višjega reda (ter tudi lambda izraze). Za ta namen uporablja vmesnik `Function<T,R>`, ki sprejme vhod tipa `T` in vrne rezultat tipa `R`. Nad funkcijo prejeto kot parameter, se kliče metoda `apply()`. Ker pa se Kotlin prevede tudi v Java 6 zložno kodo, ki funkcij višjega reda ne podpira, so znotraj paketa `kotlin.jvm.functions` deklarirani vmesniki, ki posnemajo vmesnik `Function<T, R>` iz Jave 8. Vmesnikov je točno 23, od funkcij brez argumentov do funkcij z 22-imi argumenti. Vsi vmesniki imajo funkcijo `invoke()`, katero je potrebno klicati ob dostopu Kotlin funkcij višjega reda iz Jave.

2.5.1 Vstavljene funkcije

Ker je vsaka funkcija objekt in je zanjo potrebno ustvariti objekt anonimnega razreda ter dodeliti pomnilnik, prihaja pri uporabi funkcij višjega reda do nepotrebne režije (angl. overhead). Če bi v zanki klicali funkcijo višjega

reda, kateri bi podali lambda izraz, bi se ustvarilo N novih objektov.

Vendar pa lahko v večini primerov takšne funkcije optimiziramo [4] z uporabo ključne besede `inline`, ki jo dodamo pred imenom funkcije. Ključna beseda prevajalniku pove, da vsebino klicane funkcije kopira na mesto, kjer se ta kliče. Tako ne pride do nepotrebnega ustvarjanja objektov.

Slabost vstavljenih funkcij [20] je ta, da se dolžina generirane zložne kode poveča, saj je koda funkcije sedaj na dveh mestih. Zaradi tega razloga je priporočljivo, da se s ključno besedo `inline` označi le manjše funkcije višjega reda.

2.5.2 Lambda izrazi

Lambda izraz je funkcijski literal [14]. To je funkcija, ki ni deklarirana, ni omejena na eno izmed entitet (razred, objekt ali vmesnik) in se lahko pošlje kot argument funkcijam višjega reda. Popolna sintaktična oblika lambda izraza je sledeča:

```
1  val sum = { a: Int, b: Int -> a + b }
```

Lambda izraz je obdan z zavitimi oklepaji, znotraj katerih so najprej deklarirani parametri, nato sledi puščica, za njo pa telo. Tip rezultata je avtomatsko ugotovljen na podlagi zadnjega izraza v telesu lambda izraza. Pogosteje uporabljena je sintaktična oblika, ki tipe parametrov in rezultata deklarira zunaj lambda izraza:

```
1  val sum: (Int, Int) -> (Int) = { a, b -> a + b }
```

V primeru, da ima lambda izraz le en parameter in da lahko Kotlin sam ugotovi njegov tip, se lahko do parametra dostopa preko ključne besede `it`, brez implicitne deklaracije njegovega tipa. Vrednost rezultata lambda izraza je enaka rezultatu zadnjega izraza v telesu lambda izraza. Na spodnjem primeru se vidi še ena posebnost. Če funkcija višjega reda, kot zadnji parameter prejme lambda izraz, se lahko navadni oklepaji izpustijo.

```
1  strings.find { it.length < 3 } // (it: String) -> Boolean
```

2.5.3 Anonimne funkcije

Lambda izrazom manjka možnost deklaracije tipa rezultata, ki ga funkcija vrne. V večini primerov to ni potrebno, saj prevajalnik tip ugotovi sam. Če pa to možnost potrebujemo, lahko uporabimo anonimno funkcijo (angl. anonymous function) [14]. To deklariramo na podoben način kot navadno funkcijo, le da izpustimo ime funkcije.

```
1 val sum = fun(a: Int, b: Int): Int {
2     return a + b
3 }
```

Anonimne funkcije se obnašajo podobno kot lambda izrazi, saj jih lahko pošljemo kot parameter funkciji višjega reda. Lambda izraze lahko tako nadomestimo z anonimnimi funkcijami.

2.5.4 Zaprtje funkcije

Zaprtje (angl. closure) [4] omogoča, da lambda izraz ali anonimna funkcija dostopata do spremenljivk izven njunega dosega. Tudi Java podpira zaprtja, vendar pa za razliko od Kotlina ne omogoča spreminjanja vrednosti spremenljivk.

V spodnjem primeru sta dve zaprtji. Prvo je znotraj lambda izraza funkcije `filter()`. Ta lambda izraz dostopa do vrednosti spremenljivke `x`, deklarirane zunaj obsega lambda izraza. Tukaj se vrednost le bere. Lambda izraz funkcije `forEach()`, pa dostopa do zunanje spremenljivke `sum`, kateri spreminja tudi vrednost.

```
1 fun sumOfBiggerThan(x: Int, numbers: List<Int>): Int {
2     var sum = 0
3     numbers.filter { it > x }.forEach {
4         sum += it
5     }
6     return sum
7 }
```

2.5.5 Primerjava z Javo

Tudi Java od verzije 8 najprej podpira lambda izraze [12]. Sintaksa se v primerjavi s Kotlinom malenkost razlikuje. Najprej v oklepajih sledijo argumenti funkcije, nato sledi puščica, za njo pa v zavitih oklepajih blok funkcije. Oklepaje se lahko v primeru samo enega argumenta tudi izpusti. Enako velja za telo funkcije. Če lambda izraz vsebuje le en izraz, se lahko zavite oklepaje izpusti.

```
1 BiFunction<Integer, Integer, Integer> sum = (a, b) -> {
2     Integer result = a + b;
3     return result;
4 };
```

Ker pa Kotlin podpira tudi Java 6 zložno kodo, ki lambda izrazov ne podpira, se vsak lambda izraz prevede v nov razred. Ime razreda je sestavljeno iz imena datoteke in funkcije, v kateri se lambda nahaja. Ta razred deduje abstraktni razred `Lambda` in implementira enega izmed vmesnikov `FunctionN`, kjer je N število parametrov. Razred vsebuje tudi statični primerek samega sebe, do katerega dostopa, ko se kliče lambda izraz. V primeru, da lambda izraz uporablja zaprtje, torej dostopa do spremenljivk zunaj njegovega obsega, pa generiran razred vsebuje še lastnost, ki hrani vrednost te spremenljivke. Na splošno so lambda izrazi v Kotlinu manj optimizirani kot tisti v Javi, saj se v ozadju ustvarijo dodatni razredi.

2.6 Objekti

Objekt [18] v Kotlinu ni primerek določenega razreda, temveč samostojna entiteta. Objekt je implementacija edinstvenega vzorca (angl. singleton pattern). Ta načrtovalski vzorec zagotavlja, da ima razred le en primerek, ki je skupen vsem razredom ali klientom, ki ga zahtevajo. Objekti imajo naslednje lastnosti:

- Vsebujejo lahko lastnosti, funkcije in blok `init` za inicializacijo.

- Ne smejo imeti ne primarnega ne sekundarnega konstruktorja.
- Lahko dedujejo iz ostalih razredov ali implementirajo vmesnik.

Objekt deklariramo z uporabo ključne besede `object` pred imenom objekta. Do funkcij in lastnosti objekta nato dostopamo tako, da za imenom objekta dodamo ime funkcije ali lastnosti.

```
1 object Converter {
2     fun inchToCm(inches: Double) = inches * 2.54
3 }
4 val cm = Converter.inchToCm(10.0)
```

Kotlin torej omogoča, da edinstveni vzorec ustvarimo z uporabo le ene ključne besede. V Javi je za implementacijo tega vzorca potrebno shraniti primerek razreda v privatno statično spremenljivko znotraj razreda. Razred ima nato privatni konstruktor, edini primerek razreda pa vrne javna statična metoda. In prav to v ozadju naredi Kotlin prevajalnik. Ustvari nov razred, ki ima statično lastnost `INSTANCE`, ki hrani primerek tega razreda. Zato je potrebno v primeru, da Kotlin objekt kličemo iz Jave, za imenom objekta dodati še `INSTANCE`.

```
1 double cm = Converter.INSTANCE.inchToCm(10);
```

2.6.1 Spremljevalni objekti

Ker Kotlin za razliko od Jave ne podpira statičnih funkcij in lastnosti, lahko do njih dostopamo le preko primerka razreda. Kot alternativo statičnim funkcijam in lastnostim, so na voljo spremljevalni objekti (angl. companion objects) [1]. Ti ne pripadajo primerku razreda, temveč razredu samemu. Spremljevalni objekti uporabljajo zakasnjeno inicializacijo, torej so inicializirani šele ob prvem dostopu do članov objekta. Razred, ki vsebuje spremljevalni objekt, lahko dostopa do vseh lastnosti in funkcij deklariranih v objektu, medtem ko obratno ne velja. Spremljevalni objekti lahko tudi dedujejo ali implementirajo vmesnike.

Spremljevalne objekte deklariramo znotraj razreda, označimo pa jih s ključno besedo `companion`. Do funkcij in lastnosti znotraj objekta dostopamo podobno kot do statičnih metod v Javi: najprej podamo ime razreda, nato pa ime funkcije ali lastnosti.

```
1 class Movie private constructor(val title: String, val genre:
      String) {
2     companion object {
3         fun createComedy(title: String): Movie {
4             return Movie(title, "Comedy")
5         }
6     }
7 }
8 val comedyMovie = Movie.createComedy("Airplane!")
```

V ozadju prevajalnik generira končen glavni razred (v našem primeru je to razred `Movie`), ki vsebuje notranji statičen razred `Companion`. Glavni razred vsebuje statično lastnost, ki je tipa `Companion`. Pri dostopu do lastnosti in funkcij spremljevalnega objekta iz Jave moramo eksplicitno podati tudi ime objekta.

```
1 Movie comedy = Movie.Companion.createComedy("Airplane!");
```

Če lastnosti ali funkcije spremljevalnega objekta označimo z anotacijo `@JvmStatic`, se v ozadju poleg notranjega statičnega razreda ustvarijo tudi statične metode, ki so dostopne neposredno. V tem primeru lahko pri klicu izpustimo besedo `Companion`.

2.7 Podatkovni razredi

Pri razvoju aplikacij pogosto potrebujemo razrede, ki služijo le prenosu podatkov in ne vsebujejo nobene dodatne poslovne logike. Ti razredi imajo samo lastnosti, ki hranijo vrednosti. Primer takšnih razredov so na primer razredi, ki se uporabljajo za mapiranje rezultata mrežnega klica iz formata *JSON* v objekt. Takšen razred je v Javi sestavljen iz sledečih elementov:

- spremenljivk, ki hranijo podatke,

- konstruktorja za inicializacijo razreda,
- metod, s katerimi nastavljamo in vračamo vrednosti (angl. getters and setters),
- metod `hashCode()`, `equals()` in `toString()`.

Kotlin nam omogoča, da se vsakične odvečne kode znebimo z uporabo podatkovnih razredov (angl. data class) [1]. Podatkovni razred deklariramo tako, da pred glavo razreda dodamo ključno besedo `data`. Podatkovni razredi morajo zadovoljiti naslednje zahteve:

- primarni konstruktor mora imeti vsaj en parameter,
- vsi parametri v primarnem konstruktorju morajo biti označeni z `val` ali `var`,
- podatkovni razredi ne smejo biti abstraktni, odprti (angl. open), notranji (angl. inner) ali zapečateni (angl. sealed).

Prevajalnik na podlagi podanih parametrov avtomatsko generira funkcije `equals()`, `hashCode()` in `toString()`. Razred, za katerega bi v Javi porabili okoli 50 vrstic, lahko v Kotlinu zapišemo v zgolj eni vrstici.

```
1 data class User(val firstName: String, val lastName: String,
      val username: String)
```

Poleg zgoraj omenjenih funkcij, se v ozadju generira še funkcija `copy()`, ki vrne kopijo objekta. Pri kopiranju lahko lastnostim razreda spremenimo vrednosti. V spodnjem primeru ima tako kopiran uporabnik drugačno uporabniško ime.

```
1 val user = User("Domen", "Lanisnik", "dlanisnik")
2 val copiedUser = user.copy(username = "domenl")
```

Podatkovni razredi omogočajo tudi destrukcijo objekta v posamezne spremenljivke. Za vsak atribut, ki ga deklariramo, Kotlin generira funkcijo `componentN()`, kjer N predstavlja zaporedno številko atributa in ki vrne njegovo vrednost. Tako lahko razred destruktiramo v več spremenljivk.

```
1 user.component1() // "Domen"
2 user.component2() // "Lanisnik"
3 user.component3() // "dlanisnik"
4 val (first, last, usn) = user
```

2.8 Zapečateni razredi

Zapečateni razredi (angl. sealed classes) [22] se uporabljajo za definiranje omejene množice razredov. So nekakšna razširitev naštevnih razredov, saj omogočajo, da omejimo možen nabor vrednosti, ki jih lahko spremenljivka zavzame. Zapečaten razred je abstrakten razred, torej ne more biti inicializiran neposredno. Glavni namen razreda je, da iz njega ustvarimo podrazrede, ki predstavljajo možen nabor. Zapečaten razred deklariramo s ključno besedo `sealed`, ki jo podamo pred imenom razreda. Znotraj iste datoteke deklariramo še njegove podrazrede, ki pa ne smejo uporabljati ključne besede `sealed`.

Uporabo si poglejmo na preprostem primeru aplikacije za pošiljanje sporočil. V aplikaciji lahko prejmemo več vrst vsebine: tekst, sliko in video. Vsaka vrsta vsebina ima svoje lastnosti, še vedno pa gre za vsebino sporočila. Zato lahko vsebino predstavimo kot zapečaten razred `Content`, ki ga deklariramo v datoteki `Content.kt`. V isti datoteki deklariramo še tri razrede, za vsako vrsto vsebine enega, ki dedujejo razred `Content`.

```
1 sealed class Content
2 class Text(val message: String) : Content()
3 class Image(val imageUrl: String, val caption: String) :
      Content()
4 class Video(val videoUrl: String) : Content()
```

Glavna prednost zapečatenih razredov v primerjavi z navadnim dedovanjem pride do izraza, ko jih uporabimo v izrazu `when`. Prevajalnik avtomatsko preveri, ali je pokrit ves možni nabor vrednosti in javi napako, če ni. Če je pokrit ves nabor, lahko vejo `else` izpustimo. Znotraj posamezne veje se naredi tudi samodejna pretvorba v ustrezen podrazred zapečatenega razreda.

```
1 fun displayMessageContent(content: Content) {
2     return when (content) {
3         is Text -> println(content.message)
4         is Image -> println("Image ${content.caption}: ${
              content.imageUrl}")
5         is Video -> println("Video: ${content.videoUrl}")
6     }
7 }
```

Prevajalnik v ozadju ustvari abstrakten razred, iz katerega nato dedujejo podrazredi zapečatenega razreda. V Javi je pri uporabi podrazredov potrebno s pomočjo operatorja `instanceof` preverjati, za kateri primerek razreda gre. Nato je potrebna še pretvorba v ta primerek razreda, da lahko dostopamo do njegovih lastnosti.

2.9 Zbirke

Kotlin razlikuje med dvema vrstama zbirk [23]. Nespremenljivimi, kjer so elementi deklarirani ob kreiranju zbirke in jih kasneje ne moremo spremeniti, dodati ali odstraniti, ter spremenljivimi, ki omogočajo spreminjanje zbirke.

Na vrhu razredne hierarhije zbirk je vmesnik `Iterable<out T>`, ki omogoča, da so zbirke predstavljene kot zaporedje elementov, ki podpira iteracijo. Iz njega deduje vmesnik `MutableIterable<out T>`, ki doda še funkcijo za odstranitev elementa. Nato sledi vmesnik `Collection<out E>`, ki deduje iz vmesnika `Iterable<out T>`. Ta vmesnik vsebuje splošne funkcije zbirke, kot so velikost zbirke (`size()`), ali je zbirka prazna (`isEmpty()`), ali zbirka vsebuje dan element (`contains(element: E)`) in ali zbirka vsebuje vse dane elemente (`containsAll(element: Collection<E>)`). Vmesnik se uporablja za nespremenljive sezname in množice. Za spremenljive sezname in množice pa se uporablja vmesnik `MutableCollection<E>`, ki prav tako deduje iz vmesnika `Collection<out E>` in vsebuje dodatne funkcije, kot sta naprimer funkciji za dodajanje in odstranitev elementa.

2.9.1 Seznami

Seznam ustvarimo z uporabo pomožnih funkcij, ki so na voljo v standardni knjižnici Kotlin. Najpogosteje uporabljeni funkciji sta `listOf(elements)` in `mutableListOf(elements)`. Prva ustvari nov nespremenljiv seznam iz podanih elementov, druga pa spremenljiv seznam. Na voljo je tudi funkcija `arrayListOf<T>()`, ki ustvari nov seznam javanskega tipa `ArrayList`. Za dostop do posameznega elementa seznama se tako kot pri poljih uporabljajo oglati oklepaji.

```
1 val numbers = listOf(1, 2, 3, 4)
2 numbers[2] = 10 // Ni mogoce
3 val mutableNumbers = mutableListOf(1, 2, 3, 4)
4 mutableNumbers[2] = 10
5 mutableNumbers.add(3)
```

2.9.2 Množice

Množice, podobno kot sezname, ustvarimo s pomočjo pomožnih funkcij iz standardne knjižnice Kotlin. Na voljo imamo več funkcij, ki vrnejo različne implementacije množice. Funkcija `setOf(elements)` ustvari novo nespremenljivo množico. Funkcija `hashSetOf(elements)` ustvari novo spremenljivo množico javanskega tipa `HashSet`, ki elemente hrani v razpršenem polju. Funkcija `sortedSetOf(elements)` ustvari novo spremenljivo množico javanskega tipa `TreeSet`, ki sortira elemente na podlagi njihovega naravnega vrstnega reda. Funkciji `linkedSetOf(elements)` in `mutableSetOf(elements)` pa obe ustvarita novo spremenljivo množico javanskega tipa `LinkedHashSet`, ki hrani elemente v takšnem vrstnem redu, kot so bili vstavljeni.

```
1 val set1 = setOf(8, 3, 1, 2, 6) // [8, 3, 1, 2, 6]
2 val set2 = hashSetOf(8, 3, 1, 2, 6) // [8, 1, 2, 3, 6]
3 val set3 = sortedSetOf(8, 3, 1, 2, 6) // [1, 2, 3, 6, 8]
4 val set4 = linkedSetOf(8, 3, 1, 2, 6) // [8, 3, 1, 2, 6]
5 val set5 = mutableSetOf(8, 3, 1, 2, 6) // [8, 3, 1, 2, 6]
```

2.9.3 Slovarji

Slovarji, za razliko od seznamov in množic, ne dedujejo iz nobenega vmesnika. Med drugim ponujajo funkcije za dostop do ključev in vrednosti slovarja. Slovar ustvarimo s pomočjo pomožnih funkcij iz standardne knjižnice Kotlin. Funkcija `mapOf(pairs)` ustvari nov nespremenljiv slovar iz podanih parov. Par ključ-vrednost ustvarimo z novim primerkom razreda `Pair` ali pa s pomočjo ključne besede `to` (infiksni zapis).

```
1  val map = mapOf(1 to "1", Pair(2, "2"))
```

Na voljo so tudi druge vrste slovarjev. Funkcija `mutableMapOf(pairs)` ustvari nov spremenljiv slovar. Funkcija `hashMapOf(pairs)` ustvari nov spremenljiv slovar, ki temelji na javanskem tipu `HashMap` in uporablja razpršeno polje. Funkcija `linkedHashMap(pairs)` ustvari nov spremenljiv slovar, ki temelji na javanskem tipu `LinkedHashMap` in vsebuje pare v takšnem vrstnem redu, kot so bili vstavljeni. Funkcija `sortedMapOf(pairs)` pa ustvari nov spremenljiv slovar, ki temelji na javanskem tipu `SortedMap` in hrani pare v urejenem vrstnem redu glede na ključe.

2.9.4 Operacije

Kotlin ima številne funkcije za izvajanje operacij nad zbirkami podatkov, kot so funkcije za filtriranje, sortiranje in mapiranje zbirke. Funkcije so realizirane kot razširitvene funkcije, prejmejo lambda izraz ter se lahko kličejo ena za drugo, saj kot rezultat vrnejo spremenjeno zbirko. Tako lahko v eni vrstici iz seznama oseb dobimo razvrščene naraščajoče po imenu le tiste, ki so polnoletne.

```
1  val newPersons = persons.filter { it.age >= 18 }.sortedBy {
       it.name }
```

Tudi Java od verzije 8 najprej podpira, da zbirke pretvorimo v tok podatkov (`Stream`) nad katerim lahko izvajamo operacije in na koncu tok pretvorimo nazaj v zbirko. Ker pa Kotlin podpira tudi starejše verzije Jave, so v ozadju operacije realizirane s pomočjo običajnih zank. Operacija `filter`

je tako realizirana z zanko `while`, znotraj katere se preveri, če element zbirke ustreza pogoju.

2.10 Delegirane lastnosti

Še en zelo uporaben konstrukt, ki ga ponuja Kotlin, so tako imenovane delegirane lastnosti (angl. delegated properties) [13]. Delegirana lastnost je lastnost, ki branje in pisanje vrednosti delegira drugemu razredu. Delegirano lastnost deklariramo tako, da na konec običajne lastnosti dodamo ključno besedo `by` in nato vrsto delegirane lastnosti. Nekaj vrst je vsebovanih v standardni knjižnici Kotlin, izmed katerih sta najpogosteje uporabljeni dve: `lazy` in `observable`.

Delegirana lastnost `lazy` sprejme lambda izraz, ki mora kot rezultat vrniti isti tip, kot je lastnost. Lambda se izvede šele ob prvem dostopu do lastnosti in ne ob inicializaciji objekta. Rezultat se shrani v medpomnilnik in se uporabi ob naslednjih dostopih do lastnosti. To je koristno za lastnosti, ki so računsko zahtevne in do katerih se ne dostopa v vseh primerih.

```
1 val employees: List<Employee> by lazy {
2         getAllEmployeesFromDb()
3 }
```

Delegirana lastnost `observable` sprejme dva argumenta: začetno vrednost lastnosti in lambda izraz, ki se izvede po vsaki spremembi vrednosti lastnosti. Lambda izraz prejme staro in novo vrednost lastnosti. Ta delegirana lastnost je koristna, če je ob spremembi neke lastnosti treba obvestiti ali posodobiti neko drugo lastnost.

Poglavje 3

Sistem za naročanje hrane preko mobilne aplikacije

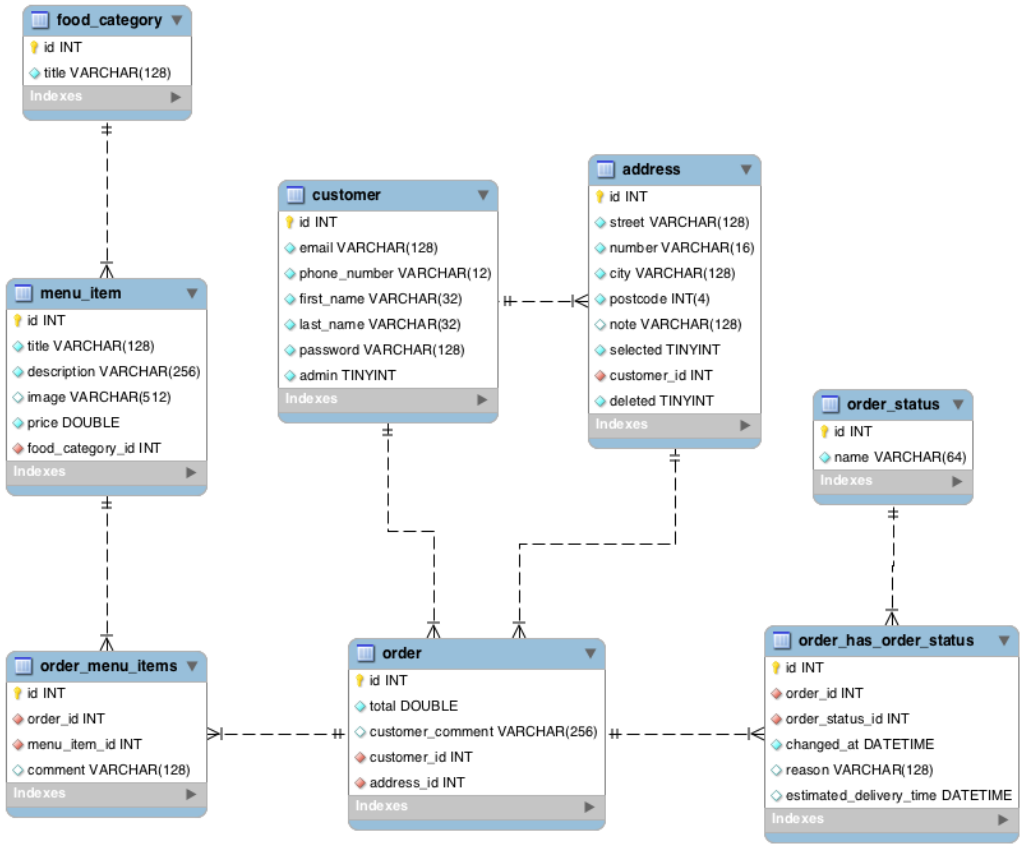
Manjše restavracije naročila še vedno sprejemajo preko telefona. Pri takšnih naročilih lahko pride do napake zaradi slabe povezave ali nerazumevanja sogovornika. Kot možno rešitev oziroma izboljšavo smo izdelali prototip sistema za naročanje hrane preko mobilne aplikacije. Sistem vključuje mobilno aplikacijo za Android, spletno aplikacijo in spletni servis, ki skupaj s podatkovno bazo povezuje mobilno in spletno aplikacijo.

Mobilna aplikacija je namenjena končnim uporabnikom. Omogoča pregled ponudbe restavracije, oddajo naročila, spremljanje statusa oddanega naročila in pregled oddanih naročil. Spletna aplikacija je namenjena zaposlenim v restavraciji in omogoča spremljanje prejetih naročil in upravljanje z njimi.

Za razvoj mobilne aplikacije, spletne aplikacije in spletnega servisa smo uporabili programski jezik Kotlin. Uporaba skupnega programskega jezika na vseh platformah nam je omogočila deljenje programske kode med njimi. Tako smo lahko podatkovne modele, ki smo jih definirali v sklopu spletnega servisa, uporabili tudi v mobilni aplikaciji in spletni aplikaciji.

3.1 Podatkovna baza

Sistem uporablja relacijsko podatkovno bazo MySQL, ki vsebuje osem tabel. Entitetno relacijski model, ki smo ga ustvarili v programu MySQL Workbench, je viden na sliki 3.1.



Slika 3.1: Entitetno-relacijski model podatkovne baze.

Tabela „customer“ vsebuje podatke o strankah in zaposlenih. Vsaka stranka je opisana z elektronsko pošto, telefonsko številko, imenom, priimkom in geslom. Zaposlenega se od stranke loči z atributom „admin“.

Stranka ima lahko shranjenih več naslovov, eden izmed naslovov pa je lahko izbran kot privzet. Ker lahko stranka v mobilni aplikaciji ureja in odstranjuje dodane naslove, se za vsak naslov hrani še logična vrednost, ki

pove, ali je bil naslov izbrisan. S tem je omogočen pregled zgodovine naročil tudi v primeru, ko je naslov naročila bil že izbrisan. Naslov lahko vsebuje tudi komentar, ki je namenjen dostavljavcu. Naslovi so predstavljeni s tabelo „address“.

Tabela „menu_item“ predstavlja jed na meniju. Vsaka jed je opisana z imenom, opisom, ceno in kategorijo, v katero spada. Dodatno ima lahko jed tudi sliko.

Osrednja tabela modela je tabela „order“, ki vsebuje vse podrobnosti oddanega naročila. Za vsako naročilo se zabeležijo podatki o stranki in izbranem naslovu za dostavo, neobvezen komentar stranke, skupen znesek in seznam izbranih jedi. Seznam izbranih jedi se hrani v ločeni tabeli, vsaki jedi pa lahko stranka doda tudi komentar. V ločeni tabeli so shranjeni tudi statusi naročila. Naročilo gre med celotnim procesom čez več statusov: ustvarjen, potrjen, zavrnjen, preklican, v izvajanju, pripravljen za dostavo in zaključen. Zabeležena je vsaka sprememba statusa naročila, skupaj s časom spremembe in razlogom.

3.2 Spletni servis

Za razvoj spletnega servisa smo uporabili integrirano razvojno okolje IntelliJ IDEA. V njem smo ustvarili nov Kotlin projekt z vključenim sistemom za izgradnjo Gradle. Spletni servis smo razvili z uporabo aplikacijskega ogrodja Spring Boot [6], ki smo ga dodali v projekt kot knjižnico. Ogrodje omogoča enostavnejši razvoj spletnih aplikacij in servisov Java, ki se lahko zaženejo s pomočjo vgrajenega strežnika Tomcat. Ogrodje ponuja tudi vrsto dodatnih knjižnic, ki pomagajo pri implementaciji pogostih zahtev, kot sta avtentikacija uporabnika in dostop do podatkovne baze. V projekt smo vključili naslednje knjižnice Spring:

- `spring-boot-starter-parent`: osnovno ogrodje za izdelavo aplikacije Spring Boot,
- `spring-boot-starter-web`: omogoča izdelavo spletnih aplikacij,

- `spring-boot-starter-security`: ogrodje za varnost aplikacije, ki vključuje avtentikacijo in avtorizacijo uporabnikov,
- `spring-boot-starter-data-jpa`: ogrodje za povezavo aplikacije s podatkovno bazo,
- `spring-boot-starter-mail`: ogrodje za pošiljanje elektronske pošte,
- `spring-boot-starter-thymeleaf`: ogrodje za izdelavo predlog Thymeleaf.

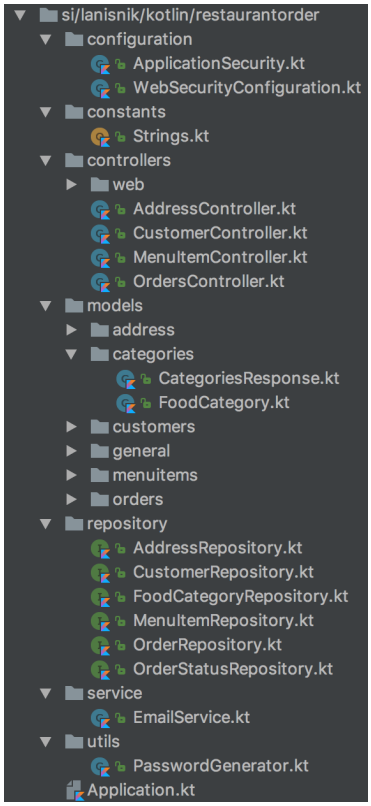
Spletni servis temelji na arhitekturi za izmenjavo podatkov med spletnimi storitvami REST (angl. Representational State Transfer). Ta predpisuje, da je vsak vir predstavljen z enoznačnim naslovom URL, dostopanje in spreminjanje vira pa je omogočeno preko standardnih metod protokola HTTP: GET, PUT, POST in DELETE.

Podatki se med spletnim servisom in mobilno aplikacijo izmenjujejo v obliki JSON (angl. JavaScript Object Notation). Prednost tega formata je preprostost in berljivost. Objekte v Javi ali Kotlinu lahko na enostaven način pretvorimo v zapis JSON in obratno.

3.2.1 Struktura projekta

Projekt je za lažji razvoj razdeljen na več logičnih delov, ki so po potrebi razdeljeni še naprej. S tem so razredi z istimi funkcionalnostmi združeni skupaj, kot je prikazano na sliki 3.2. Na osnovnem nivoju se nahaja datoteka `Application.kt`, ki je potrebna za konfiguracijo in zagon aplikacije. V paketu `configuration` se nahajajo razredi za konfiguracijo varnosti spletne aplikacije. Paket `constants` vsebuje datoteke s konstantami, ki se uporabljajo čez celotno aplikacijo. Krmilniki aplikacije, ki so vstopna točka spletnega servisa, se nahajajo v paketu `controllers`. Znotraj paketa `models` se nahajajo podatkovni modeli, ki so združeni glede na uporabo. Repozitoriji za dostop do podatkovne baze in izvedbo operacij nad podatki, se nahajajo v paketu `repository`. V paketu `service` se nahajajo dodatni pomožni servisi,

kot je servis za pošiljanje elektronske pošte. V paketu `utils` pa se hranijo dodatni pomožni razredi.



Slika 3.2: Struktura projekta za spletni servis.

3.2.2 Dostop do podatkovne baze

Dostop do podatkovne baze smo implementirali s programskim vmesnikov za podatkovne baze v Javi (angl. Java Persistence API) in objektno-relacijskim preslikovalnim mehanizmom (angl. object-relational mapping) Hibernate. JPA [2] je javanska specifikacija, ki opisuje delo z relacijskimi podatkovnimi bazami v aplikacijah Java. Ker gre le za specifikacijo, je za sam dostop in shranjevanje objektov v podatkovno bazo potreben še ORM. Ta omogoča preslikanje podatkov iz podatkovne baze v običajne objekte Java (angl. Plain

Old Java Object).

Za vzpostavitev povezave med aplikacijo in podatkovno bazo, je bilo potrebno urediti datoteko `application.properties`, ki vsebuje nastavitve aplikacije. Dodali smo naslov, na katerem je dostopna baza, ime sheme ter uporabniško ime in geslo za dostop.

Naslednji korak je bila izdelava podatkovnih modelov za preslikovanje iz podatkovne baze v objekte. Spodnji izsek kode prikazuje primer, kako smo s pomočjo anotacij, ki jih definira JPA, deklarirali entiteto za tabelo jedi. Z anotacijo `@Entity` povemo, da razred predstavlja tabelo v podatkovni bazi, ime tabele pa je enako imenu razreda. To je privzeto mapirno pravilo, ki ga lahko v nastavitvah aplikacije po želji spremenimo. Z anotacijo `@Id` označimo primarni ključ entitete, anotacija `@GeneratedValue` pa določa pravilo, po katerem se primarni ključ generira. V konkretnem primeru se izbere najbolj primerno pravilo glede na nastavitve podatkovne baze. Anotacijo `@ManyToOne` uporabimo za predstavitev relacije med dvema tabelama, z `@JoinColumn` pa povemo na podlagi katerega atributa naj se izvede združitev.

```
1  @Entity
2  class MenuItem{
3      @Id
4      @GeneratedValue(strategy = GenerationType.AUTO)
5      val id: Int = 0
6      val title: String = ""
7      val description: String = ""
8      val image: String? = null
9      val price: Double = 0.0
10     @ManyToOne
11     @JoinColumn(name = "food_category_id")
12     val category: FoodCategory? = null
13 }
```

Nato smo za vse entitete definirali še repozitorije. Repozitorij [9] je enostaven vmesnik, ki definira operacije nad določeno entiteto. Definirali smo jih kot razširitev generičnega vmesnika `CrudRepository<T, ID>`. Ta je del

Spring Boota in kot vhod sprejme tip entitete in podatkovni tip primarnega ključa. S tem so omogočene osnovne operacije CRUD za branje, ustvarjanje, spreminjanje in brisanje nad podano entiteto. Svoje operacije definiramo tako, da funkciji pripišemo še dodatno anotacijo s stavkom SQL. V primeru poizvedovalnih operacij to ni potrebno, saj se poizvedba generira na podlagi podpisa funkcije.

Spodaj je zapisan primer repozitorija za jedi. Poleg osnovnih operacij CRUD smo definirali še operacijo, ki poišče vse jedi, ki pripadajo podani kategoriji.

```
1 interface MenuItemRepository : CrudRepository<MenuItem, Int>
      {
2     fun findByCategoryId(categoryId: Int): List<MenuItem>
3 }
```

Deklaracija vmesnika repozitorija je vse, kar je potrebno za izvajanje operacij nad podatkovno bazo. Spring Boot namreč ob zagonu aplikacije ustvari implementacijo deklariranega repozitorija.

3.2.3 Varnost

Varnost aplikacije [5] smo vklopili tako, da smo ustvarili konfiguracijski razred `ApplicationSecurity` in mu dodali anotacijo `@EnableWebSecurity`. Znotraj razreda smo definirali varnostna pravila za dostop do posameznih storitev. Tako je dostop do nekaterih storitev možen le za avtenticirane uporabnike.

Spletni servis uporablja osnovno avtentikacijo (angl. basic authentication), kjer se uporabniško ime in geslo zakodirata z binarno kodo Base64. Rezultat kodiranja se pošlje v polju `Authorization`, v glavi protokola HTTP. Pri uporabniškem imenu „Domen“ in geslu „Geslo“, bi bila vsebina polja `Basic RG9tZW46R2VzbG8=`. Na strežniku se vrednost polja dekodira, geslo pa se primerja s tistim v podatkovni bazi. V primeru, da se gesli ne ujemata, strežnik vrne odgovor nepooblaščeno (angl. unauthorized) s statusno kodo 401, dostop do storitve pa je onemogočen.

Dodatno so nekatere storitve dostopne le uporabnikom, ki imajo vlogo `ADMIN`, tisti z vlogo `USER` pa do njih ne morejo dostopati. S tem je zagotovljeno, da imajo do administratorskega dela spletne aplikacije dostop le zaposleni in ne vsak registriran uporabnik.

Preverjanje pravic uporabnika se izvaja znotraj konfiguracijskega razreda `WebSecurityConfiguration`. Ves opisani postopek pa samodejno ob klicu vsake storitve opravi Spring Boot.

3.2.4 Krmilniki

Krmilnik (angl. controller) je vstopna točka vsakega klica spletne storitve. Gre za razred, ki je anotiran z anotacijo `@RestController` in vsebuje funkcije, ki predstavljajo servise. Funkcije morajo deklarirati metodo HTTP in naslov URL, preko katerega bo servis dostopen. To je omogočeno s pomočjo anotacije `@RequestMapping`. Na podlagi klicanega naslova URL se izvede ustrezna funkcija znotraj ustreznega krmilnika.

Krmilniki lahko poleg funkcij vsebujejo tudi repozitorije za dostop do podatkovne baze. Ker gre pri repozitorijih le za vmesnike, je potrebno spremenljivke označiti z anotacijo `@Autowired`. Spring Boot nato vstavi dejansko implementacijo.

Storitve za dostop do seznama jedi in kategorij smo deklarirali v krmilniku `MenuItemController` (glej tabelo 3.1). Krmilnik vsebuje dva repozitorija, enega za jedi in drugega za kategorije ter dve storitvi. Obe storitvi sta dosegljivi preko metode GET in brez avtentikacije.

Storitev `/menu-items/categories` vrne seznam vseh kategorij, ki so shranjene v podatkovni bazi.

Druga storitev `/menu-items/category` pa vrne seznam vseh jedi za izbrano kategorijo. Identifikator izbrane kategorije se pošlje kot parameter imenovan `id` v naslovu URL.

Storitve za urejanje uporabnikovih naslovov za dostavo so deklarirane v

Opis	Naslov URL	Metoda
Seznam kategorij	/menu-items/categories	GET
Seznam jedi za določeno kategorijo (ID kategorije kot parameter)	/menu-items/category	GET

Tabela 3.1: Pregled storitev za jedi in kategorije

krmilniku `AddressController`. Krmilnik vsebuje štiri storitve (tabela 3.2), za vse pa velja, da so na voljo le avtenticiranim uporabnikom. Storitve uporabljajo dva repozitorija, enega za delo z uporabniki, drugega pa za delo z naslovi.

Na osnovnem naslovu URL `/address` sta dosegljivi dve storitvi, ena preko metode GET, druga pa preko metode PUT. Prva vrne seznam vseh naslovov za dostavo, ki so v podatkovni bazi shranjeni za avtenticiranega uporabnika, ki kliče to storitev. Druga pa v podatkovno bazo shrani nov naslov za dostavo. Podatki o novem naslovu so poslani v telesu zahtevka HTTP, kot objekt `AddressRequest`.

Tretja storitev uporabniku omogoča izbris izbranega naslova za dostavo. Dosegljiva je preko metode DELETE, kot del poti pa vsebuje identifikator naslova. V ozadju se naslov ne izbriše iz podatkovne baze, temveč se le označi kot izbrisan. Tako so podatki o naslovu vidni tudi pri naročilih, ki so imela izbran izbrisan naslov.

Zadnja storitev je dosegljiva preko metode POST in označi naslov, katerega identifikator je poslan kot del poti, kot privzet. V podatkovni bazi se za ta naslov nastavi vrednost `selected` na 1, vsem drugim naslovom uporabnika pa na 0. S tem je pri oddaji naročila v mobilni aplikaciji ta naslov avtomatsko izbran.

Krmilnik `CustomerController` vsebuje storitve za upravljanje z uporabniki (tabela 3.3). Prve tri storitve so dosegljive preko metode POST in brez avtentikacije.

Opis	Naslov URL	Metoda
Seznam naslovov za uporabnika	/address	GET
Dodaj nov naslov (Naslov v obliki JSON v telesu zahtevka)	/address	PUT
Odstrani naslov	/address/{id}	DELETE
Izberi naslov kot privzet	/address/{id}/select	POST

Tabela 3.2: Pregled storitev za urejanje uporabnikovih naslovov

Storitev za prijavo prejme v telesu zahtevka objekt `LoginRequest`, ki vsebuje naslov elektronske pošte in geslo uporabnika. Najprej se preveri, če je uporabnik s to elektronsko pošto shranjen v podatkovni bazi, nato pa še, če se prejeto in shranjeno geslo ujemata. V primeru, da sta oba pogoja zadoščena, se kot rezultat vrne profil uporabnika (objekt `Customer`), v nasprotnem primeru pa napaka.

Storitev za registracijo prejme v telesu zahtevka objekt `RegisterRequest` s podatki novega uporabnika: ime in priimek, naslov elektronske pošte, telefonska številka in geslo. Pred shranjevanjem novega uporabnika v podatkovno bazo se najprej preveri, da uporabnik s to elektronsko pošto še ne obstaja. Geslo se v podatkovno bazo shranjuje kot zgoščena vrednost dolžine 60 znakov, ki je pridobljena kot rezultat zgoščevalne funkcije *bcrypt* [29]. Značilnost te zgoščevalne funkcije je, da algoritem vedno znova generira naključen niz, ki ga doda vhodu (angl. salt). Ta je v Spring Bootu implementirana kot del pomožne knjižnice. Kot rezultat uspešne registracije se vrne profil novo ustvarjenega uporabnika.

Storitev za ponastavitev gesla prejme objekt `ResetPasswordRequest`, ki vsebuje naslov elektronske pošte. Uporabniku s to elektronsko pošto se v podatkovni bazi zamenja geslo z novim, naključno generiranim geslom. Novo geslo je nato uporabniku poslano preko elektronske pošte. Za pošiljanje sporočil je zadolžen servis `EmailService`.

Zgoraj opisane storitve so dosegljive brez avtentikacije, medtem ko je le-ta potrebna za ostale storitve znotraj tega krmilnika. Spremembe v podatkovni

Opis	Naslov URL	Metoda
Prijava	/customer/login	POST
Registracija	/customer/register	POST
Ponastavitev gesla	/customer/resetPassword	POST
Podatki o stranki	/customer/profile	GET
Sprememba gesla	/customer/changePassword	POST
Sprememba podatkov	/customer/updateProfile	POST

Tabela 3.3: Pregled storitev za uporabnika

bazi se izvedejo nad avtenticiranim uporabnikom, ki te storitve kliče.

Storitev `/customer/profile`, ki je dosegljiva preko metode GET, vrne uporabnikove podatke znotraj objekta `Customer`. To storitev uporablja mobilna aplikacija za pridobitev najbolj svežih podatkov stranke, saj se lahko podatki vrnjeni ob prijavi ali registraciji, v vmesnem času spremenijo. To se lahko zgodi v primeru, ko se uporabnik prijavi v aplikacijo na drugi napravi in spremeni svoje ime.

Zadnji dve storitvi sta dosegljivi preko metode POST, potrebne podatke pa prejmeta v telesu zahtevka. Prva storitev omogoča spremembo trenutnega gesla uporabnika. Storitev prejme trenutno geslo in novo geslo. Pred zapisom novega gesla se najprej preveri, če se prejeto trenutno geslo ujema s tistim v podatkovni bazi. Nato se to geslo v podatkovni bazi povozi z zgoščeno vrednostjo novega gesla. Druga storitev omogoča spremembo uporabnikovih osebnih podatkov. Prejeti ime in priimek, naslov elektronske pošte in telefonska številka se posodobijo v podatkovni bazi. Kot rezultat se vrne posodobljen profil uporabnika.

Storitve povezane z naročili se nahajajo v krmilniku `OrdersController` (tabela 3.4). Najpomembnejša storitev je `/order/create`, ki je dostopna preko metode POST. V telesu zahtevka prejme objekt `CreateOrderRequest`, ki vsebuje identifikator izbranega naslova za dostavo, opcijski komentar stranke in seznam izbranih jedi. Na podlagi teh podatkov se v podatkovno bazo

Opis	Naslov URL	Metoda
Oddaja novega naročila	/order/create	POST
Seznam vseh naročil za stranko	/order/history	GET
Podrobnosti naročila	/order/history/{id}	GET

Tabela 3.4: Pregled storitev za naročila

zapiše novo naročilo s statusom „ustvarjen“.

Seznam vseh oddanih naročil za uporabnika, se pridobi s klicem storitve `/order/history`. Vrnjen seznam naročil je urejen padajoče, po času zadnje spremembe statusa. Tako so v mobilni aplikaciji novejša naročila prikazana na vrhu zaslona. Ta storitev vrne le nekatere informacije o posameznem naročilu, kot so identifikator, znesek za plačilo, zadnji status in število naročenih jedi. Podrobnosti o posameznem naročilu pa vrača servis `/order/history/{id}`. Te podrobnosti med drugim vsebujejo seznam vseh sprememb statusov, podatke o naslovu za dostavo in seznam izbranih jedi. Ker bi lahko z ugibanjem identifikatorja naročila videli podrobnosti tudi o naročilu drugega uporabnika, se ob klicu storitve najprej preveri, da se avtenticiran uporabnik ujema z uporabnikom, ki je zapisan na tem naročilu.

3.2.5 Testiranje

Spletni servis smo testirali sproti, tako da smo po vsaki spremembi lokalno pognali projekt in nato spremenjene spletne servise klicali s pomočjo programa Postman. Če je servis vrnil napako ali pa se ni obnašal kot pričakovano, smo izvajanje spletnega servisa zaustavili in ga po odpravi napake ponovno pognali. Dodatno smo spletni servis testirali in nadgrajevali še pri razvoju mobilne aplikacije.

3.3 Mobilna aplikacija

Mobilno aplikacijo za operacijski sistem Android smo razvili v integriranem razvojnem okolju Android Studio 3. Aplikacijo smo v celoti napisali v programskem jeziku Kotlin. Poleg Kotlina smo uporabili še XML za zaslonske maske in Gradle za konfiguracijo projekta. Aplikacija je namenjena verzijam operacijskega sistema Android 5.0 (Lollipop) in novejšim.

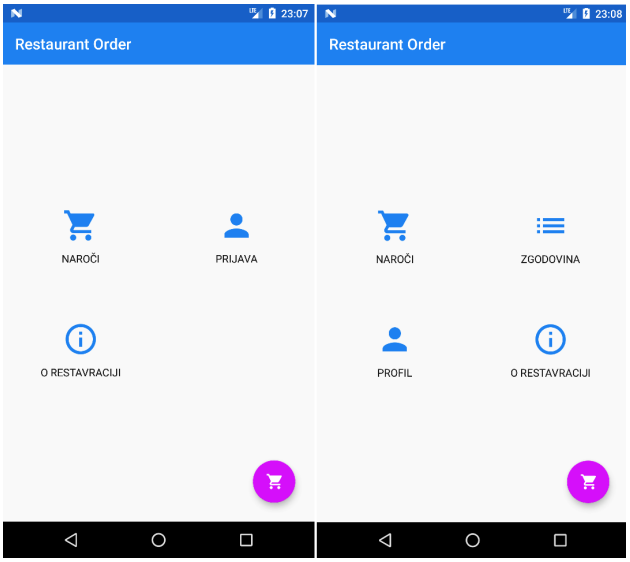
3.3.1 Delovanje aplikacije in zaslonske maske

Ob zagonu aplikacije se odpre nadzorna plošča, ki je tudi glavna vstopna točka aplikacije. Nadzorna plošča vsebuje gumbe za dostop do ostalih delov aplikacije. Prikazani gumbi se razlikujejo glede na tip uporabnika, kar prikazuje slika 3.3. Neprijavljeni uporabniki vidijo le tri gumbe: „Naroči“, „Prijava“ in „O restavraciji“. Prijavljeni uporabniki pa vidijo štiri gumbe: „Naroči“, „Zgodovina“, „Profil“ in „O restavraciji“. Dinamična postavitev gumbov je dosežena z uporabo vizualnega gradnika `GridView`, ki elemente prikaže v mreži.

Prijava v aplikacijo

Klik na gumb „Prijava“ odpre zaslon za prijavo. Zaslon vsebuje obrazec za prijavo in povezave do zaslonov za registracijo in ponastavitev gesla. Vsi trije zasloni so prikazani na sliki 3.4. Obrazec za prijavo je sestavljen iz vnosnega polja za elektronsko pošto, vnosnega polja za geslo in gumba za izvedbo prijave. Klik na gumb za prijavo izvede validacijo vnesenih podatkov. V primeru manjkajočega podatka ali nepravilne oblike elektronske pošte se prikaže obvestilo. V primeru, da so vneseni podatki pravilni, se pokliče spletni servis za prijavo. Če ta vrne uspešen odgovor, se vrnjene uporabnikove podatke shrani lokalno. Na koncu se naredi preusmeritev nazaj na nadzorno ploščo, ki sedaj prikazuje vse štiri gumbe.

Če uporabnik še nima računa, ga lahko ustvari na zaslonu za registracijo.



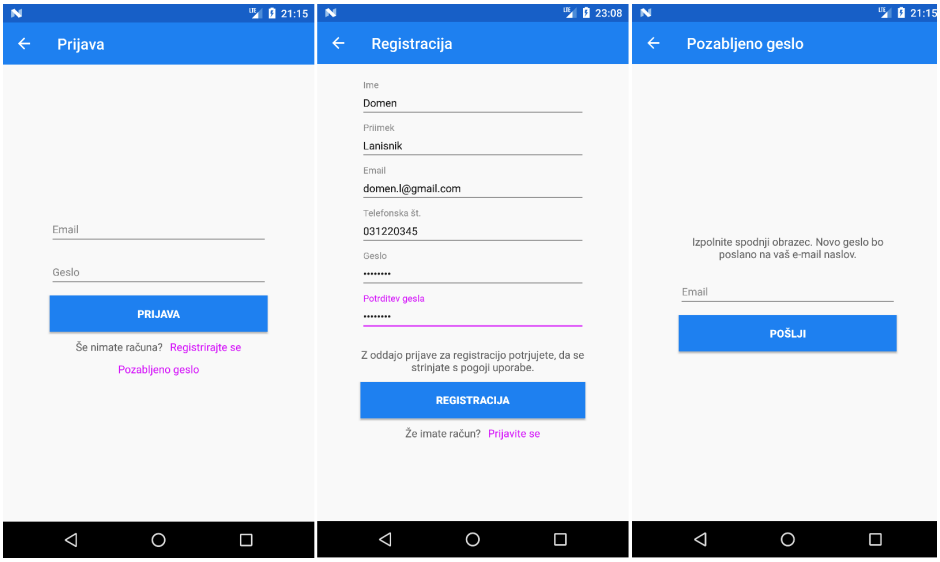
Slika 3.3: Nadzorna plošča za neprijavljenega in prijavljenega uporabnika.

Obrazec je sestavljen iz vnosnih polj za ime, priimek, elektronsko pošto, telefonsko številko, geslo in potrditev gesla. Enako kot pri prijavi, se tudi tukaj pred pošiljanjem vnesenih podatkov na spletni servis izvede validacija. Dodatno se preveri še, da se obe vneseni gesli ujemata. Po uspešni registraciji se lokalno shranijo uporabnikovi podatki in uporabnika se preusmeri nazaj na nadzorno ploščo.

V primeru, da uporabnik že ima račun, vendar pa je pozabil geslo, si ga lahko ponastavi s pomočjo obrazca za pozabljeno geslo. Obrazec je sestavljen iz vnosnega polja za elektronsko pošto in gumba „Pošlji“. Po validaciji vnesenih podatkov se kliče spletni servis za ponastavitev gesla. Ob uspešnem odgovoru se zaslon zapre in ponovno je viden zaslon za prijavo. Novo geslo je uporabniku poslano po elektronski pošti na vnesen naslov.

Uporabnikov profil

Po prijavi uporabnika, se na nadzorni plošči gumb „Prijava“ zamenja z gumbom „Profil“. Klik na ta gumb odpre zaslon, kjer uporabnik ureja svoje osebne podatke (slika 3.5).



Slika 3.4: Obrazci za prijavo, registracijo in ponastavitev gesla.

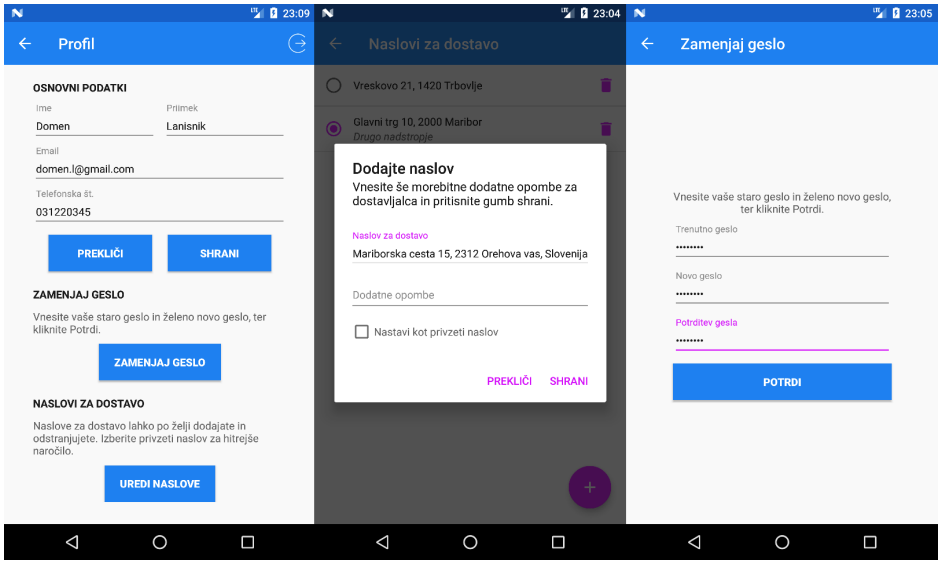
Gumb za odjavo se nahaja v desnem kotu orodne vrstice zaslona. Ob odjavi se iz lokalne podatkovne baze odstranijo vsi shranjeni podatki. Učinek je tako isti, kot če bi aplikacijo zagnali prvič.

Na vrhu zaslona se nahaja obrazec za spremembo osebnih podatkov. Vnosna polja so predizpolnjena s trenutnimi podatki uporabnika. Ob kliku na gumb „Shrani“, se spremembe validirajo in nato pošljejo na spletni servis.

Preko gumba „Zamenjaj geslo“ je dostopen zaslon za menjavo gesla. Zaslon vsebuje obrazec za vnos trenutnega gesla, novega gesla in potrditev novega gesla. Podatki se po validaciji vnosa pošljejo na spletni servis.

Gumb „Uredi naslove“ odpre zaslon za urejanje naslovov za dostavo. Obstoječi naslovi so prikazani v seznamu. Vsaka celica v seznamu ima dva gumba: enega za izbiro tega naslova kot privzetega, ter enega za odstranitev naslova. Ob kliku na gumba se spremembe posredujejo spletnemu servisu in seznam se osveži. Uporabnik doda nov naslov s klikom na lebdeči gumb v desnem spodnjem kotu zaslona. Odpre se dialog za iskanje naslova preko storitve Google, kjer uporabnik vnese svoj naslov. Iz seznama rezultatov izbere svoj naslov, ki se nato prikaže v novem dialogu. Uporabnik lahko doda

še neobvezen komentar in izbere ali naj bo nov naslov izbran kot privzet. V primeru, da klikne na gumb „Shrani“, se podatki o novem naslovu pošljejo na spletni servis, drugače se postopek dodajanja prekine.

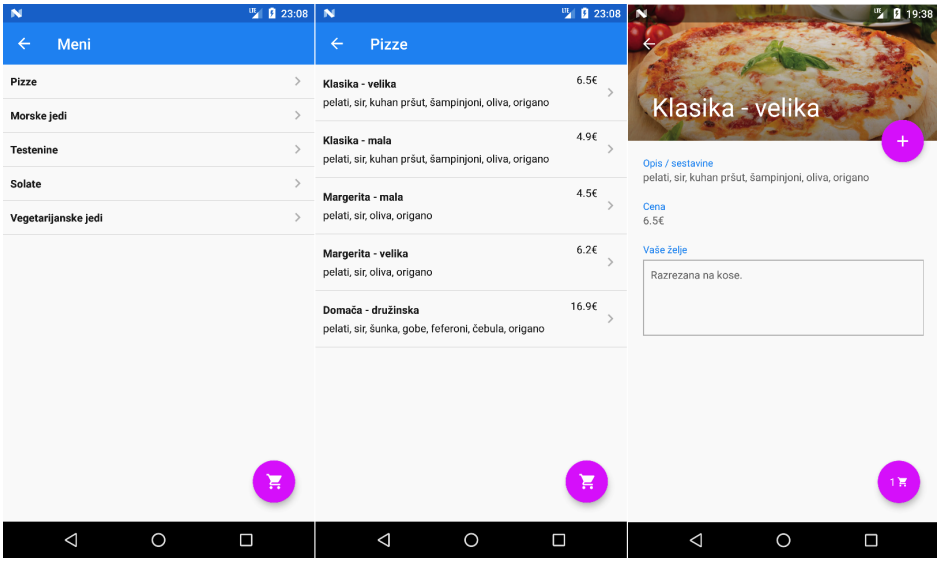


Slika 3.5: Zaslon za urejanje osebnih podatkov, zaslon za urejanje naslovov in zaslon za spremembo gesla.

Izbira jedi

Postopek oddaje novega naročila se začne s klikom na gumb „Naroči“ na nadzorni plošči. Odpre se zaslon s seznamom kategorij jedi, ki so na voljo. Zaslon je prikazan na sliki 3.6. Klik na eno izmed kategorij odpre nov zaslon, ki prikazuje seznam vseh jedi znotraj izbrane kategorije. Za vsako jed so prikazani naslednji podatki: naziv jedi, opis jedi in cena v evrih. Klik na jed odpre nov zaslon s podrobnostmi izbrane jedi.

Na vrhu zaslona se nahajata slika in naziv jedi, pod sliko pa opis jedi, cena jedi in vnosno polje. V vnosno polje lahko uporabnik vnese svoje želje oziroma komentar glede jedi. Zaslon vsebuje tudi dva akcijska gumba. Gumb na vrhu zaslona doda trenutno jed k naročilu, gumb na dnu zaslona pa odpre podrobnosti naročila.



Slika 3.6: Zaslon za izbiro kategorije jedi, zaslon za izbiro jedi in zaslon s podrobnostmi izbrane jedi.

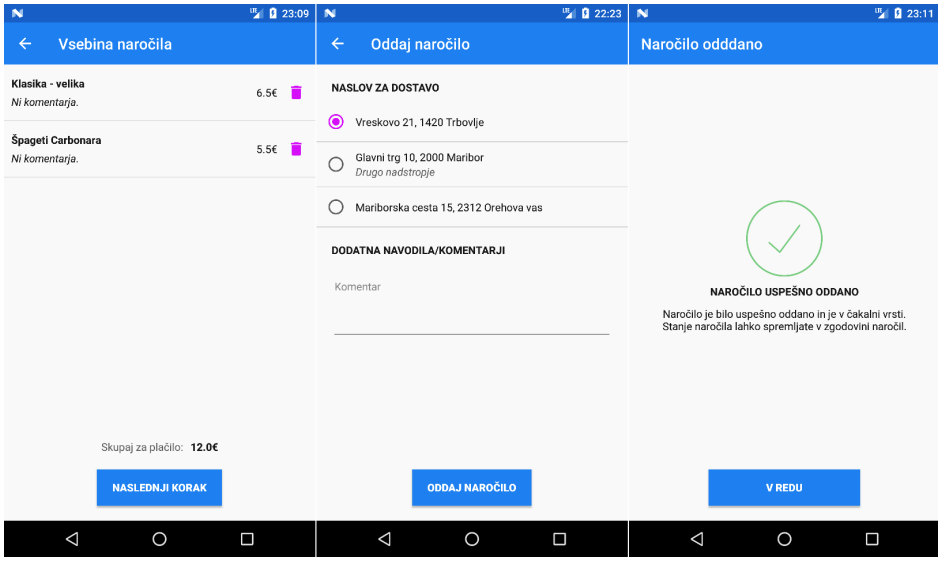
Gumb za dostop do podrobnosti naročila je viden na nadzorni plošči, seznamu kategorij, seznamu jedi in podrobnostih jedi. Na gumbu je zapisana tudi številka, ki odraža število naročenih jedi. Ko se k naročilu doda nova jed, se na vseh zgoraj naštetih zaslonih posodobi tudi gumb. Stanje naročila se shranjuje v lokalni podatkovni bazi. Če torej dodamo k naročilu tri jedi in nato zapremo aplikacijo, se ob ponovnem zagonu aplikacije ohranita tako stanje naročila kot tudi število na gumbu.

Oddaja naročila

Zaslon s podrobnostmi naročila prikazuje seznam dodanih jedi. Za vsako jed so prikazani naziv jedi, cena jedi, uporabnikov komentar k jedi in gumb za odstranitev jedi iz naročila. Pod seznamom se nahaja skupni znesek za plačilo in gumb za prehod na naslednji korak postopka naročila. V primeru, da ni bilo dodane še nobene jedi, sta skupni znesek in gumb za naslednji korak skrita, prikaže pa se napis „Vaše naročilo je prazno.“.

Klik na gumb za naslednji korak odpre zaslon, kjer uporabnik izbere

naslov za dostavo in vnese morebiten komentar k naročilu. V primeru, da uporabnik nima dodanega še nobenega naslova za dostavo, mu je oddaja naročila onemogočena. Postopek oddaje naročila je zaključen ob kliku na gumb „Oddaj naročilo“. Podatki o naročilu so poslani na spletni servis, kjer se ustvari novo naročilo. Ob uspešnem odgovoru se prikaže zaslon, kjer se uporabnika obvesti, da je bilo naročilo uspešno oddano in da lahko stanje naročila spremlja znotraj aplikacije. Ob zaprtju zaslona o uspehu se ponovno prikaže nadzorna plošča. Vsi trije zasloni so vidni na sliki 3.7.

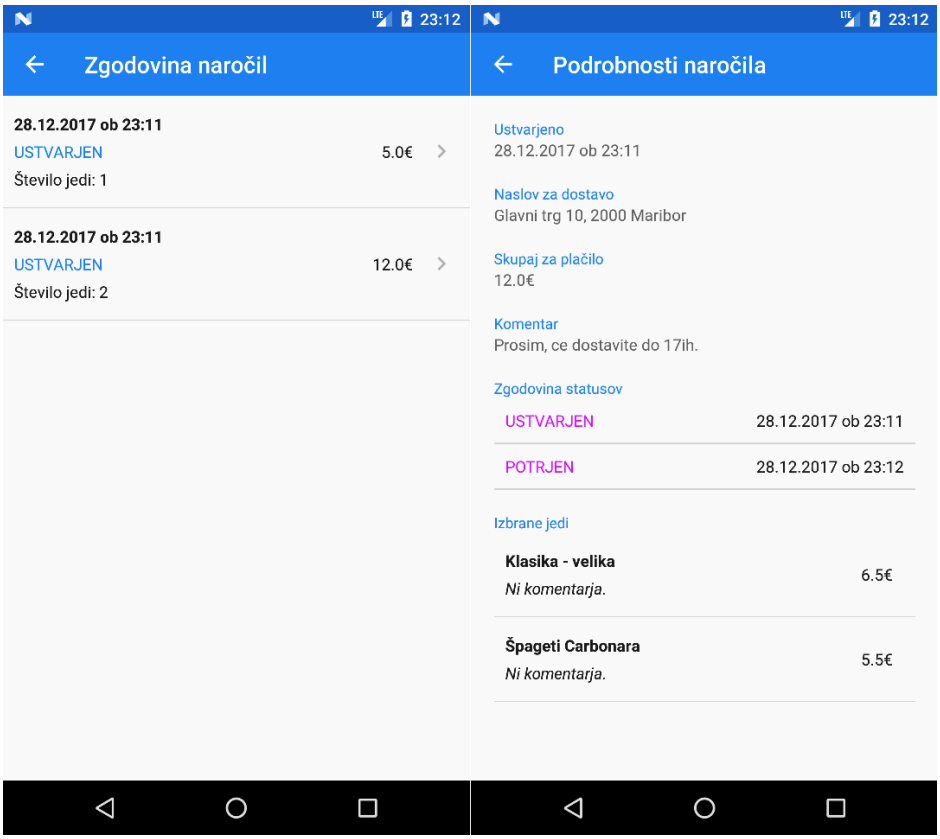


Slika 3.7: Zaslon za pregled naročila, zaslon za oddajo naročila in zaslon ob uspešno oddanem naročilu.

Zgodovina naročil

Pomemben del aplikacije je tudi zgodovina naročil, ki je dostopna iz nadzorne plošče. Kot je vidno na sliki 3.8, zaslon prikazuje seznam vseh oddanih naročil uporabnika. Naročila so urejena padajoče po datumu zadnje spremembe statusa. To pomeni, da je najbolj aktualno naročilu na vrhu seznama oziroma zaslona. Za vsako naročilo je prikazan trenuten status, datum zadnje spremembe statusa, skupni znesek in število jedi. Ob izbiri naročila se odpre nov

zaslon z njegovimi podrobnostmi. Zaslon prikazuje seznam sprememb statusov naročila, seznam izbranih jedi, izbran naslov za dostavo, skupen znesek plačila in komentar k naročilu.



Slika 3.8: Zaslona za pregled zgodovine oddanih naročil.

Ostalo

Zadnji zaslon, ki je dosegljiv iz nadzorne plošče, pa je zaslon „O restavraciji“. V trenutni verziji aplikacije prikazuje le tekst z opisom aplikacije.

3.3.2 Uporabljene knjižnice

Zaradi lažjega razvoja in večje nadgradljivosti aplikacije smo v projekt vključili tudi dodatne knjižnice. Te smo dodali z uporabo sistema za izgradnjo Gradle,

na enak način, kot smo to storili pri spletnem servisu. V tem podpoglavju so opisane nekatere ključne knjižnice.

Retrofit

Retrofit [25] je klient HTTP, ki poenostavi izmenjavo podatkov s spletnim servisom. Zahtevke HTTP deklariramo v Java vmesniku. Posamezen zahtevek opišemo z uporabo anotacij, s katerimi povemo naslov URL, metodo HTTP, tip rezultata, parametre in podobno. Retrofit podpira uporabo pretvornikov, ki odgovor v formatu JSON pretvorijo v objekt z istimi lastnostmi. Klient omogoča izvajanje sinhronih in asinhronih klicev.

Realm

Realm [24] je NoSQL podatkovna baza, ki je optimizirana za mobilne naprave. Podatki so predstavljeni z objekti, ki jih definiramo in nad katerimi lahko izvajamo povpraševanja. Prednosti pred podatkovno bazo SQLite so enostavna konfiguracija, enostavna uporaba in hitrost.

Android Architecture Components

Android Architecture Components [10] je zbirka več knjižnic, razvitih s strani Googla. V našem projektu uporabljamo osnovno knjižnico, ki vsebuje komponenti `ViewModel` in `LiveData`. Komponenti sta ključni pri arhitekturi aplikacije.

RxJava

RxJava [26] je knjižnica za reaktivno programiranje na platformi JVM. Omogoča izdelavo programov, ki asinhrono opazujejo podatkovne tokove in se odzivajo na dogodke ali spremembe podatkov. Knjižnica vključuje tudi operatorje za združevanje več podatkovnih tokov v en sam tok. Glavna prednost knjižnice je ta, da sama poskrbi za upravljanje z nitmi in sinhronizacijo med njimi.

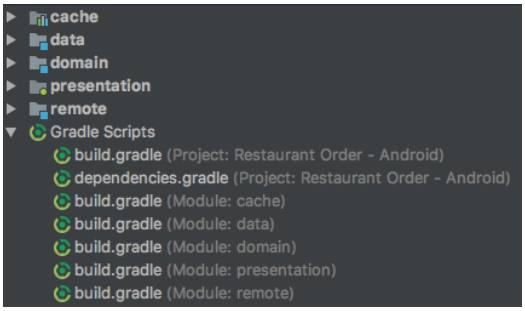
Dagger

Dagger [28] je ogrodje za vstavljanje odvisnosti (angl. dependency injection). Temelji na anotacijah, na podlagi katerih generira kodo za dostop do lastnosti razreda. Ključne anotacije so `@Module`, `@Provides`, `@Inject` in `@Component`. Razredi označeni z anotacijo `@Module` so ponudniki objektov, ki so lahko vstavljeni. Takšni razredi vsebujejo funkcije označene z anotacijo `@Provides`. Anotacija `@Component` se uporablja na vmesnikih, ki povezujejo module. Dagger na podlagi teh vmesnikov izvede vstavljanje potrebnih odvisnosti. Odvisnosti so označene z anotacijo `@Inject`, katero se lahko uporabi na konstruktorju, funkciji ali posamezni lastnosti.

3.3.3 Arhitektura

Aplikacija je zasnovana po principu čiste arhitekture (angl. clean architecture) [11]. Ta princip razdeli aplikacijo v več nivojev, za katere velja, da notranji nivoji ne smejo vedeti za zunanje nivoje. Komunikacija med nivoji poteka preko vmesnikov. Na ta način je programska koda neodvisna od specifičnih ogrodij in knjižnic, lažje vzdržljiva in lažja za testiranje.

Aplikacijo smo razdelili v pet nivojev, kot je to prikazano na sliki 3.10. Nivoje smo realizirali kot module oziroma knjižnice znotraj projekta. Struktura projekta v Android Studiu je prikazana na sliki 3.9.



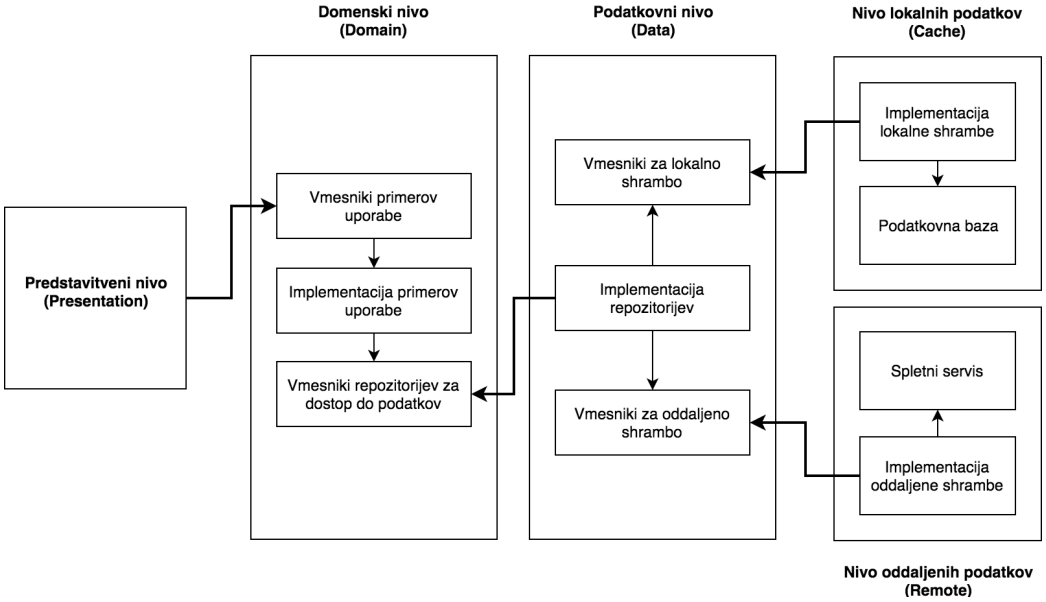
Slika 3.9: Struktura projekta v Android Studiu.

Najbolj notranji in najpomembnejši nivo je domenski nivo. Ta vsebuje vso poslovno logiko aplikacije in je popolnoma neodvisen od ostalih nivojev in od operacijskega sistema Android. Domenski nivo definira podatkovne modele, deklarira vmesnike repozitorijev za dostop do podatkov in implementira primere uporabe. Primer uporabe predstavlja določeno funkcionalnost aplikacije, običajno je to pridobivanje podatkov iz repozitorijev. Primer takšnega primera uporabe je `LoginCustomer`, ki izvede prijavo uporabnika v aplikacijo. Z vhodnimi podatki (elektronska pošta in geslo) kliče funkcijo deklarirano v vmesniku za repozitorij `CustomerRepository` in kot rezultat vrne uspeh ali napako. Do repozitorija dostopa asinhrono s pomočjo knjižnice RxJava.

Naslednji nivo je podatkovni nivo, ki se nahaja nad domenskim nivojem in je od njega odvisen. Podatkovni nivo implementira repozitorije deklarirane v domenskem nivoju. Odgovoren je za dostop do lokalne in oddaljene shrambe, ki jih definirana z vmesniki. Podatkovni nivo deklarira lastne podatkovne modele in pretvornike za pretvorbo v modele domenskega nivoja. Prej omenjena funkcija za prijavo uporabnika v aplikacijo je implementirana znotraj razreda `CustomerDataRepository`. Funkcija naprej dostopa do oddaljene shrambe in ob uspešnem odgovoru shrani pridobljene podatke v lokalno shrambo.

Nivo lokalnih podatkov se nahaja nad podatkovnim nivojem. Implementira lokalno shrambo, ki je deklarirana v podatkovnem nivoju. Odgovoren je za dostop do lokalne podatkovne baze oziroma kakšne druge vrste shrambe. Znotraj nivoja se uporabljajo posebni podatkovni modeli, ki vsebujejo tudi anotacije potrebne za delovanje knjižnice Realm. Podatkovni modeli se nato pretvorijo v modele podatkovnega nivoja. Ker je implementacija podatkovne baze omejena le na ta nivo, lahko trenutno knjižnico za podatkovno bazo zamenjamo s kakšno drugo. Pri tem pa ni treba spremeniti ostalih nivojev aplikacije.

Nivo oddaljenih podatkov, tako kot nivo lokalnih podatkov, se nahaja nad podatkovnim nivojem. Implementira oddaljeno shrambo deklarirano v podatkovnem nivoju. Odgovoren je za izmenjavo podatkov s spletnim servisom REST. Tudi ta nivo uporablja svoje podatkovne modele, ki se nato pretvorijo v modele podatkovnega nivoja. Tako je možno zamenjati klienta za izvajanje klicev HTTP, ne da bi to vplivalo na druge dele aplikacije.



Slika 3.10: Arhitektura mobilne aplikacije.

Najbolj zunanji nivo je predstavitveni nivo, ki vključuje tudi vse ostale nivoje. Predstavitveni nivo vsebuje vso specifično programsko kodo Android, kot so aktivnosti (angl. activity), zaslonske maske, prehodi med zasloni in odzivanje na uporabnikovo interakcijo z aplikacijo. Predstavitveni nivo uporablja arhitekturo MVVM oziroma „model-pogled-model pogleda“ (angl. Model-View-ViewModel) [3]. Model pogleda (angl. ViewModel) predstavlja posrednika med modelom in pogledom. Dostopa do modela, ki hrani podatke, in dobljene podatke posreduje pogledu. Pogled opazuje lastnosti v modelu pogleda in ob spremembah upodobi novo stanje. V našem primeru je pogled predstavljen z aktivnostjo, model pa s primerom uporabe.

3.3.4 Testiranje

Pomemben del razvoja mobilne aplikacije je tudi testiranje. Aplikacijo smo med razvijanjem sproti testirali v emulatorju, ki je del programskega okolja Android Studio. Emulator posnema delovanje različnih naprav Android. To nam je omogočilo, da smo aplikacijo preizkusili na zaslonih različnih velikosti.

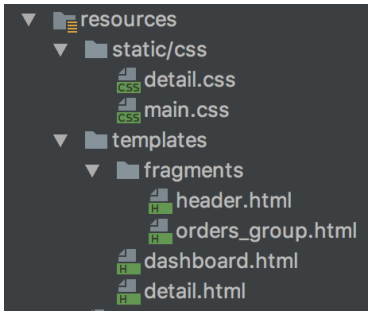
3.4 Spletna aplikacija

Spletno aplikacijo smo izdelali znotraj istega projekta kot spletni servis. Razdeljena je na dva dela: zaslonskih mask in zalednega dela. Za razvoj smo poleg Kotlina uporabili še označevalni jezik HTML (angl. Hypertext Markup Language) in slogovni jezik CSS (angl. Cascading Style Sheets). Z njima smo razvili zaslonske maske aplikacije. Pri tem nam je bila v pomoč knjižnica Thymeleaf [27], s katero smo izdelali predloge HTML. Predloge omogočajo uporabo spremenljivk, kontrolnih stavkov in zank znotraj elementov HTML. Vrednosti spremenljivk so nato posredovane s strani krmilnika, ki je odgovoren za prikaz predloge. Pred upodobitvijo spletne strani so spremenljivke zamenjane z njihovimi vrednostmi.

Spodnji primer izraza prikazuje preprosto uporabo predlog. Ob upodobitvi spletne strani se vsebina značke `span` zamenja z vrednostjo spremenljivke `comment`, oziroma poševnico, če ima spremenljivka ničelno vrednost.

```
1 <span th:text="${comment != null} ? ${comment} : '/'"></span>
```

Pri izdelavi zaslonskih mask smo uporabili tudi spletno razvojno ogrodje Bootstrap. S tem smo na enostaven način dosegli enoten izgled elementov in samodejno prilagajanje postavitve elementov velikosti zaslona.



Slika 3.11: Struktura zaslonskih mask spletne aplikacije.

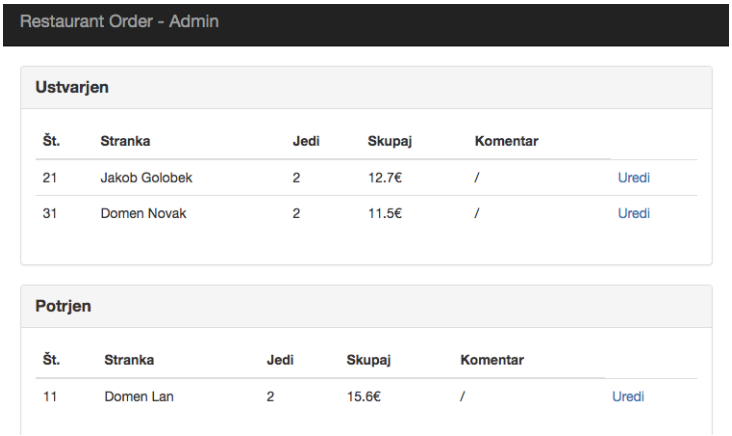
3.4.1 Delovanje

Ob odprtju spletne aplikacije se najprej prikaže sistemski dialog, ki poziva k vnosu uporabniškega imena in gesla zaposlenega. Po uspešni avtentikaciji se odpre nadzorna plošča (slika 3.12), kjer so prikazana vsa prejeta naročila. Naročila so grupirana po njihovih trenutnih statusih. Za vsako naročilo je prikazan identifikator naročila, ime in priimek stranke, število naročenih jedi, skupni znesek in morebiten komentar stranke. Za podrobnejši pregled naročila ali spremembo njegovega statusa, je potreben klik na gumb „Uredi“, ki odpre podrobnosti. Na zaslonu s podrobnostmi naročila so prikazani podatki o stranki, naslovu za dostavo, izbranih jedeh in zgodovini statusov. Na dnu strani je dodaten okvir, ki vsebuje vnosni obrazec za spremembo statusa naročila. Uporabnik s spustnega seznama izbere nov status, vnese neobvezen razlog za spremembo, ter klikne na gumb „Potrdi“. Po kliku je preusmerjen nazaj na nadzorno ploščo, kjer se prikažejo spremembe.

3.4.2 Zaslonske maske

Datoteke z zaslonskimi maskami se nahajajo v posebni mapi oziroma paketu `resources`, kot je vidno na sliki 3.11. Aplikacija je sestavljena iz dveh glavnih zaslonskih mask. Obe sta predlogi knjižnice Thymeleaf in izpisujeta vrednosti spremenljivk, ki jih prejmeta od krmilnika.

Prva je `dashboard.html`, ki predstavlja nadzorno ploščo in vstopno točko



Slika 3.12: Nadzorna plošča spletne aplikacije.

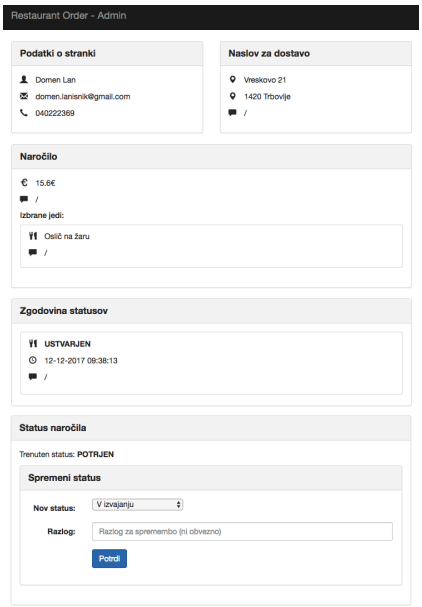
aplikacije. Zaslonska maska je sestavljena iz več fragmentov. Fragment je ločena predloga, ki jo lahko vključimo v drugo predlogo HTML. V našem primeru je ta fragment deklariran v datoteki `orders_group.html` in ima nalogo izpisati ime podanega statusa ter izrisati tabelo podanih naročil. Fragmenti se dodajo dinamično glede na prejete statuse v spremenljivki `statuses`. Fragment prejme seznam naročil (objekti `Order`) v spremenljivki `orders`. S pomočjo ukaza iz knjižnice Thymeleaf (glej sliko 3.13), se za vsako naročilo dinamično generira nova vrstica v tabeli. Vsaka vrstica je sestavljena iz šestih stolpcev: številke naročila, imena stranke, števila naročenih jedi, zneska za plačilo, komentarja in gumba, ki odpre podrobnosti naročila.

```
<tr th:each="order: ${orders}" >
    <td th:text="${order.id}">1</td>
    <td th:text="${order.customer.firstName} + ' ' + ${order.customer.lastName}">Janez Novak</td>
    <td th:text="${#lists.size(order.menuItems)}">3</td>
    <td th:text="${order.total} + '€'">10€</td>
    <td th:text="${order.customerComment != null} ? ${order.customerComment} : '/'">Komentar</td>
    <td>
        <a href="#"
           th:href="@{/{id}(id=${order.id})}">Uredi</a>
    </td>
</tr>
```

Slika 3.13: Del kode zaslonske maske, kjer se generira vsebina tabele naročil.

Druga zaslonska maska je `detail.html`, ki prikazuje podrobnosti izbra-

nega naročila (slika 3.14). Sestavljena je iz več okvirjev (razred `panel` iz knjižnice Bootstrap), ki prikazujejo podatke o stranki, izbranem naslovu za dostavo, izbranih jedeh in zgodovini statusov. Na dnu se nahaja vnosni obrazec (element `form`) za spremembo statusa naročila. Obrazec je sestavljen iz spustnega seznama (element `select`), vnosnega polja (element `input`) in gumba za potrditev (element `button`). Ob kliku na gumb se vnos shrani in pošlje preko metode POST krmilniku, ki prevzame izvajanje.



Slika 3.14: Podrobnosti izbranega naročila v spletni aplikaciji.

3.4.3 Zaledni del

Ustvarili smo nov krmilnik `AdminDashboardController` zadolžen za upodobitev prej definiranih predlog. Znotraj krmilnika smo definirali funkcije, ki enako kot v primeru ostalih krmilnikov, predstavljajo servise. Razlikujejo se le v tem, da kot parameter prejmejo vmesnik `Model`, vrnejo pa pot do datoteke HTML, ki naj se prikaže. Vmesnik `Model` omogoča nastavljanje vrednosti spremenljivkam, ki so uporabljene v predlogah.

Krmilnik vsebuje štiri storitve. Dostop do njih je omogočen le avtenticiranim uporabnikom z vlogo administratorja. Prva storitev je dosegljiva na osnovnem naslovu in naredi preusmeritev na drugo storitev, to je `/dashboard`. Ta storitev je odgovorna za prikaz pregledne plošče administratorskega dela. Iz repozitorija za naročila najprej pridobi seznam vseh naročil. Naročila so nato grupirana glede na njihov zadnji status. Grupirana naročila so nato nastavljena kot atribut `orders` v vmesniku `Model`. Kot rezultat se vrne pot datoteke `dashboard.html`.

Tretja storitev se nahaja na naslovu `/id`, kjer `id` predstavlja identifikator izbranega naročila. Storitev se kliče ob kliku na gumb uredi in prikaže podrobnosti izbranega naročila (datoteka `detail.html`). Iz repozitorija za naročila se na podlagi identifikatorja pridobi objekt `Order`. Vmesniku `Model` se nato nastavijo atributi, ki so potrebni za prikaz vseh podrobnosti. Najpomembnejši so:

- `customer` - objekt `Customer`, za prikaz podatkov o stranki,
- `address` - objekt `Address`, za prikaz podatkov o izbranem naslovu za dostavo,
- `items` - seznam objektov `OrderMenuItems`, za prikaz izbranih jedi,
- `statuses` - seznam objektov `OrderStatusChanges`, urejenih padajoče po času spremembe,
- `changeStatusRequest` - objekt `ChangeStatusRequest`, v katerega se zapiše uporabnikov vnos, ko uporabnik spremeni status naročila.

V primeru, da zaposleni spremeni status naročila (klik na gumb „Potrdi“), se pošlje objekt `ChangeStatusRequest` kot vhodni parameter zadnji storitvi. Ta je dosegljiva na istem naslovu (`/id`), vendar preko metode POST. Naročilu nastavi nov status, shrani spremembo v podatkovno bazo in preusmeri uporabnika nazaj na nadzorno ploščo (storitev `/dashboard`), kjer je vidna tudi zadnja sprememba.

3.5 Testiranje

Poleg sprotnega testiranja spletnega servisa in mobilne aplikacije ob samem razvoju smo na koncu testirali še delovanje celotnega sistema. Šli smo skozi celoten postopek prijave v mobilno aplikacijo in oddaje novega naročila. Nato smo preverili še spletno aplikacijo, da pravilno prikazuje novo ustvarjena naročila in da so spremembe statusa naročila vidne tudi v mobilni aplikaciji. Zaznane napake smo odpravljali sproti.

Po končanem lokalnem testiranju smo se odločili, da testiranje mobilne aplikacije omogočimo še ožjemu krogu uporabnikov. Spletni servis in aplikacijo smo zato morali namestiti na oddaljen spletni strežnik, mobilno aplikacijo pa popraviti, da se je povezala na novi naslov URL. Zaradi drugačnega okolja spletnega strežnika, je bilo treba popraviti oziroma dopolniti nekatere dele spletnega servisa in spletne aplikacije. Za testiranje vsake spremembe je bilo potrebno aplikacijo Spring Boot ponovno zgraditi in jo naložiti na spletni strežnik. Ko smo odpravili vse napake, smo zgradili še mobilno aplikacijo in končno datoteko `.apk` poslali drugim v test. Na podlagi oddanih naročil, ki so jih izvedli v mobilni aplikaciji, smo testirali še spletno aplikacijo za zaposlene.

Rezultat testiranja mobilne aplikacije s strani uporabnikov je bil seznam popravkov, ki pa so bili predvsem vizualne narave: napačna postavitev teksta, prekrivanje elementov in podobno. Po odpravi napak smo uporabnikom ponovno poslali aplikacijo v test. V novi verziji aplikacije ni bilo odkritih novih napak, s čimer smo testiranje zaključili.

Poglavje 4

Zaključek

V diplomskem delu smo naprej predstavili nov sodobni programski jezik Kotlin. Na podlagi kratkih primerov smo predstavili njegove osnovne in naprednejše konstrukte. Ogledali smo si, kako do nekaterih konstruktov dostopamo iz Jave in kako so ostali konstrukti, ki jih Java ne podpira ali pa so omejeni na novejše verzije jezika, implementirani v Kotlinu.

Prehod iz Jave v Kotlin je bil na začetku malo težji, predvsem zaradi drugačne sintakse nekaterih osnovnih konstruktov jezika. Po nekaj dneh uporabe Kotlina smo se jezika dokončno navadili. K temu je pripomoglo tudi razvojno okolje IntelliJ IDEA, ki nudi odlično podporo jeziku in daje priporočila za bolj berljivo kodo. Med samo uporabo Kotlina nismo imeli nobenih težav z združljivostjo knjižnic, ki so napisane v Javi. Jezik se je izkazal za stabilnega in primernega za vsakodnevno uporabo. Za najbolj uporabne konstrukte so se izkazali podatkovni razredi in razširitvene funkcije. S pomočjo podatkovnih razredov smo lahko v eni vrstici programske kode opisali podatkovni model, z razširitvenimi funkcijami pa smo si bistveno pohitrili razvoj, saj smo pogoste operacije, kot je naprimer skrivanje komponente uporabniškega vmesnika, skrajšali na samo en klic funkcije.

V praktičnem delu smo v Kotlinu razvili sistem za naročanje hrane preko mobilne aplikacije za Android, sestavljen iz spletnega servisa, spletne apli-

kacije in mobilne aplikacije. S sistemom smo želeli ponuditi rešitev tistim restavracijam, ki naročila še vedno sprejemajo preko telefona. Menimo, da naša rešitev olajša postopek naročila, tako strankam, kot tudi sami restavraciji. Mobilna aplikacija je enostavna za uporabo in vsebuje vse pomembnejše funkcionalnosti, kot je prijava v aplikacijo, pregled ponudbe, urejanje naslovov za dostavo, urejanje osebnih podatkov, enostavna oddaja naročila, pregled oddanih naročil in spremljanje statusa naročila. Spletna aplikacija pa zaposlenim v restavraciji omogoča enostaven pregled nad naročili in sledenje izvajanju naročila.

Menimo, da ima rešitev, ki smo jo razvili potencial in bi jo lahko z nekaj nadgradnjami še izboljšali. Na podlagi odzivov uporabnikov bi k izboljšanju uporabniške izkušnje največ prinesla potisna sporočila. Uporabnik bi bil tako o vsaki spremembi stanja naročila obveščen v istem trenutku. Pri naročilu jedi bi bilo smiselno uporabniku ponuditi tudi možnost izbora dodatkov. Trenutno to funkcionalnost nadomešča vnosno polje. Ena izmed ključnih funkcionalnosti, ki bi sistem naredila še bolj uporaben za restavracije, pa bila mobilna aplikacija namenjena dostavljavcem. Ta bi dostavljavcem omogočala načrtovanje poti za razvoz naročil in nudila pregled nad naročili.

Programski jezik Kotlin ponuja večino prednosti, ki so značilne za Javo, obenem pa naslavlja njene večje pomanjkljivosti. Jezik se razvija še naprej in je deležen posodobitev, ki izboljšujejo optimiziranost jezika in prinašajo nove sodobne konstrukte. Zaradi odlične kompatibilnosti je prehod iz Jave na Kotlin zelo enostaven, sam prehod pa se lahko zgodi postopoma. Njegova popularnost, predvsem pri razvoju aplikacij za operacijski sistem Android, še naprej raste. Vse več člankov in tečajev za razvoj mobilnih aplikacij namesto Jave uporablja Kotlin. Upamo, da bo naše diplomsko delo v pomoč razvijalcem, ki se želijo naučiti novega sodobnega jezika z ogromnim potencialom.

Literatura

[1] Antonio Leiva. *Kotlin for Android Developers*. Leanpub, 2017.

[2] Bojan Brumen. Java persistance API (JPA) ter hibernate predmetno-relacijski preslikovalni mehanizem. Diplomska naloga, Fakulteta za elektrotehniko, računalništvo in informatiko, Univerza v Mariboru, 2011.

[3] Dejan Obrez. Ogrodje mobilne aplikacije mFRI. Diplomska naloga, Fakulteta za računalništvo in informatiko, Univerza v Ljubljani, 2017.

[4] Dmitry Jemerov and Svetlana Isakova. *Kotlin in Action*. Manning Publications, 2017.

[5] Felipe Gutierrez. Pro Spring Boot (chapter: Security with Spring Boot, pages 177–209). Springer, 2016.

[6] K. S. Prasad Reddy. Beginning Spring Boot 2 (chapter: Spring Boot Essentials, pages 47–53). Springer, 2017.

[7] Manish Jangid. Kotlin–the unrivalled android programming language lineage. *Imperial Journal of Interdisciplinary Research*, vol. 3, no. 8, pp. 256–259, 2017.

[8] Mirjam Založnik. Izbrani koncepti programskega jezika kotlin. Diplomska naloga, Fakulteta za elektrotehniko, računalništvo in informatiko, Univerza v Mariboru, 2016.

[9] Accessing data with JPA. Dosegljivo: `https://spring.io/guides/gs/accessing-data-jpa/`, 2017. [Dostopano: 20. 12. 2017].

[10] Android architecture components. Dosegljivo: https://developer.android.com/topic/libraries/architecture/index.html, 2017. [Dostopano: 27. 12. 2017].

[11] Android clean architecture components boilerplate. Dosegljivo: https://github.com/bufferapp/clean-architecture-components-boilerplate, 2017. [Dostopano: 22. 12. 2017].

[12] Java - lambda izrazi. Dosegljivo: https://docs.oracle.com/javase/tutorial/java/javaOO/lambdaexpressions.html, 2017. [Dostopano: 16. 11. 2017].

[13] Kotlin - delegirane lastnosti. Dosegljivo: https://kotlinlang.org/docs/reference/delegated-properties.html, 2017. [Dostopano: 3. 11. 2017].

[14] Kotlin - funkcije najvišjega nivoja, lambda izrazi. Dosegljivo: https://code.tutsplus.com/tutorials/kotlin-from-scratch-more-functions--cms-29479, 2017. [Dostopano: 16. 11. 2017].

[15] Kotlin - osnovni tipi. Dosegljivo: https://kotlinlang.org/docs/reference/basic-types.html, 2017. [Dostopano: 2. 12. 2017].

[16] Kotlin - pogosta vprašanja. Dosegljivo: https://kotlinlang.org/docs/reference/faq.html, 2017. [Dostopano: 14. 12. 2017].

[17] Kotlin - razredi. Dosegljivo: https://kotlinlang.org/docs/reference/classes.html, 2017. [Dostopano: 10. 11. 2017].

[18] Kotlin - razredi in objekti. Dosegljivo: https://code.tutsplus.com/tutorials/kotlin-from-scratch-classes-and-objects--cms-29590, 2017. [Dostopano: 10. 11. 2017].

[19] Kotlin - varnost pred ničelnimi vrednostmi. Dosegljivo: https://kotlinlang.org/docs/reference/null-safety.html, 2017. [Dostopano: 28. 10. 2017].

[20] Kotlin - vstavljene funkcije. Dosegljivo: https://kotlinlang.org/docs/reference/inline-functions.html, 2017. [Dostopano: 15. 11. 2017].

[21] Kotlin - zanke in odločitveni stavki. Dosegljivo: https://kotlinlang.org/docs/reference/control-flow.html, 2017. [Dostopano: 3. 12. 2017].

[22] Kotlin - zapečateni razredi. Dosegljivo: https://kotlinlang.org/docs/reference/sealed-classes.html, 2017. [Dostopano: 18. 11. 2017].

[23] Kotlin - zbirke. Dosegljivo: https://code.tutsplus.com/tutorials/kotlin-from-scratch-ranges-and-collections--cms-29397, 2017. [Dostopano: 1. 11. 2017].

[24] Realm database. Dosegljivo: https://realm.io/products/realm-database, 2017. [Dostopano: 27. 12. 2017].

[25] Retrofit. Dosegljivo: http://square.github.io/retrofit, 2017. [Dostopano: 27. 12. 2017].

[26] Rxjava: Reactive extensions for the JVM. Dosegljivo: https://github.com/ReactiveX/RxJava, 2017. [Dostopano: 27. 12. 2017].

[27] Thymeleaf. Dosegljivo: http://www.thymeleaf.org, 2017. [Dostopano: 22. 12. 2017].

[28] Using dagger 2 for dependency injection in android - tutorial. Dosegljivo: http://www.vogella.com/tutorials/Dagger/article.html#special-treatment-of-fields-in-dagger, 2017. [Dostopano: 27. 12. 2017].

[29] Zgoščevalna funkcija bcrypt. Dosegljivo: https://en.wikipedia.org/wiki/Bcrypt, 2017. [Dostopano: 20. 12. 2017].